

Noțiuni de bază (I)

După cum am văzut, în textul sursă al unui program C sunt descrise datele utilizate de program și acțiunile care au loc asupra lor în timpul rulării programului. Compilatorul organizează alocarea datelor în memorie și traduce acțiunile (funcțiile) în cod mașină, iar link-editorul asociază programului nostru diverse alte funcții gata compilate. La execuție, sistemul de operare alocă programului o anumită zonă de memorie în care încarcă codurile funcțiilor și valorile inițiale ale datelor, după care lansează funcția **main**. Execuția programului constă într-o serie de transferuri de octeți între memorie și regiștrii procesorului, într-o mișcare înainte și înapoi cu o frecvență foarte ridicată. Octeții datelor ajung în regiștrii aritmetici unde sunt prelucrați conform instrucțiunilor - care și ele sunt copiate rând pe rând și octet cu octet din memorie în registrul de instrucțiuni al procesorului. Pentru ca toată această mișcare de octeți să aibă coerență și finalitate, programatorul trebuie să definească de la bun început structura datelor folosite și să precizeze clar modul lor de prelucrare, altfel spus, trebuie să declare pentru fiecare dată *tipul* ei.

1. Tipuri fundamentale

Tipul unei date precizează mărimea și organizarea locației de memorie în care este stocată data respectivă, precum și comportamentul datei sub acțiunea operatorilor. De exemplu, dacă **x** și **y** sunt două variabile de tip întreg cu valorile 7 și respectiv 2, atunci expresia **x/y** are ca rezultat numărul întreg 3, iar dacă **x** și **y** sunt variabile în virgulă mobilă cu aceleași valori numerice, rezultatul evaluării expresiei **x/y** este 3,5.

Precizarea tipului unei date se face în mod explicit printr-o *declarație de tip*, care trebuie să apară în program înaintea oricărei utilizări a datei respective.

Tipurile pot fi predefinite (aparținând limbajului) sau pot fi definite de utilizator. Următoarele tipuri predefinite sunt fundamentale: tipul **void**, tipul logic **bool**, tipurile aritmetice întregi **char** și **int** și tipurile aritmetice în virgulă mobilă **float** și **double**. Toate celelalte tipuri (tablouri, funcții, pointeri, structuri, clase, etc) sunt *tipuri derivate* plecând de la aceste tipuri fundamentale.

Tipul **void** este utilizat pentru a declara funcții care nu returnează nimic (rezultat de tip **void**), sau care nu au argumente formale (lista argumentelor formale este de tip **void**). Celelalte utilizări ale tipului **void** privesc în special variabilele de tip **pointer** și vor fi precizate la momentul oportun.

Tipul logic **bool** a fost introdus în limbaj odată cu trecerea la C++, are numai două valori reprezentate de cuvintele cheie **true** și **false**, pentru păstrarea compatibilității cu operațiile logice din limbajul C tipul **bool** se comportă în calcule ca un tip aritmetic, memorat pe un octet, cu valoarea 1 pentru **true** și 0 pentru **false**.

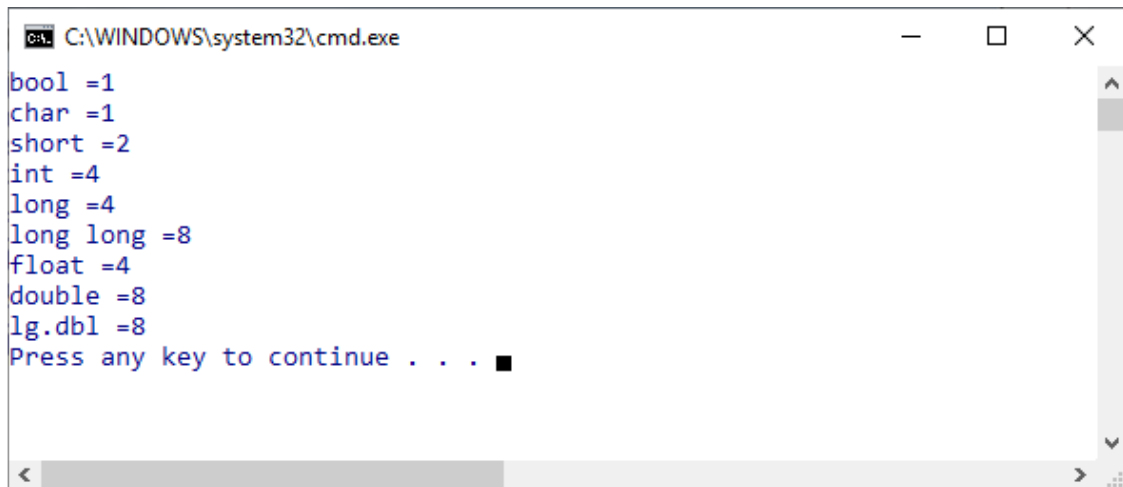
Tipurile aritmetice pot fi prefixate de *modificatorii de tip short/long* (care precizează mărimea spațiului de alocare) și de modificatorii **signed/unsigned** (care afectează domeniul de valori), obținându-se în final 14 tipuri aritmetice distincte. Fiecare tip poate avea și denumiri sinonime, fiind preferată forma cea mai scurtă de scriere care asigură distincția între tipuri:

- 1) **char**;
- 2) **signed char**;
- 3) **unsigned char**;
- 4) **short** = short int = signed short = signed short int;
- 5) **unsigned short** = unsigned short int;
- 6) **int** = signed int = signed;
- 7) **unsigned** = unsigned int;
- 8) **long** = long int = signed long = signed long int;
- 9) **unsigned long** = unsigned long int;
- 10) **long long** = long long int = signed long long int
- 11) **unsigned long long** = unsigned long long int
- 12) **float**;
- 13) **double**;
- 14) **long double**.

Spațiul de alocare și domeniul de valori al fiecărui tip aritmetic depind de implementare, limbajul C garantând numai anumite incluziuni (cu posibilitate de egal) între domeniile de valori ale tipurilor aritmetice. Astfel, pentru întregi ordinea de incluziune este: **char**, **short**, **int**, **long**, iar pentru numere raționale: **float**, **double**, **long double**. Implementarea acestor tipuri în MS Visual C++ este dată în fișierul `tipuri_aritmetice.pdf`.

Programatorul poate afla mărimea spațiului de stocare al fiecărui tip de dată cu ajutorul operatorului **sizeof** care, aplicat unui anumit tip, întoarce ca rezultat numărul de octeți ocupați de o variabilă de acel tip. Vezi exemplului următor.

```
int main(){
    cout << "bool =" << sizeof(bool) << endl;
    cout << "char =" << sizeof(char) << endl;
    cout << "short =" << sizeof(short) << endl;
    cout << "int =" << sizeof(int) << endl;
    cout << "long =" << sizeof(long) << endl;
    cout << "long long =" << sizeof(long long) << endl;
    cout << "float =" << sizeof(float) << endl;
    cout << "double =" << sizeof(double) << endl;
    cout << "lg.dbl =" << sizeof(long double) << endl;
    return 0;
}
```

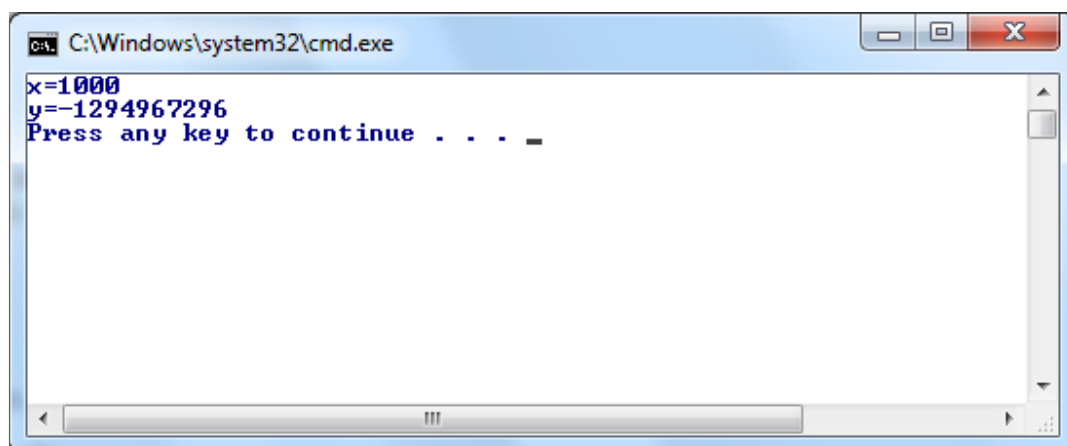


```
C:\WINDOWS\system32\cmd.exe
bool =1
char =1
short =2
int =4
long =4
long long =8
float =4
double =8
lg.dbl =8
Press any key to continue . . . █
```

Cunoașterea domeniului de valori a unei variabile este foarte importantă, deoarece depășirea domeniului nu este semnalizată în nici un fel și conduce sigur la rezultate eronate. Programul

```
int main() {
    int x=1000;
    int y=3*x*x*x;
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
    return 0;
}
```

are următorul rezultat surprinzător:



```
C:\Windows\system32\cmd.exe
x=1000
y=-1294967296
Press any key to continue . . . █
```

2. Aritmetica tipului **char**

Tipul caracter (**char**, **signed char** și **unsigned char**) este utilizat în special pentru scrierea și citirea textelor, conform unei codificări numerice a setului de caractere disponibil, codificare dependentă de compilator și de sistemul de operare folosit (MS Visual C++ utilizează codul ASCII cu extinderea “IBM® character set”).

Deoarece variabilele de tip caracter sunt de fapt numere întregi reprezentate pe un octet, ele se supun, în consecință, aritmeticii modulo 256. Pentru sublinierea acestui aspect fundamental al datelor de tip caracter vom prezenta folosirea acestora în calcule aritmetice, deși această utilizare a lor este secundară. Regula de bază este următoarea: la evaluarea unei expresii aritmetice orice dată caracter este promovată automat la tipul **int** conform domeniului său de valori și este utilizată valoarea numerică obținută în urma acestei promovări. De reținut că atribuirile de forma „tip caracter trece în tip întreg” păstrează valoarea numerică numai dacă valoarea caracterului este în domeniul de valori al tipului întreg considerat, în caz contrar, de exemplu la o atribuire de forma „valoare negativă trece în **unsigned int**”, are loc o schimbare de semnificație a formatului intern de reprezentare a datelor, situație care trebuie evitată. Atribuirea inversă, de forma „tip întreg trece în tip caracter” se face totdeauna prin pierderea celor mai semnificativi octeți (deci prin trecerea la clase de resturi modulo 256), situație care iarăși trebuie evitată.

Tipul **signed char** are domeniul de valori de la -128 până la 127 iar **unsigned char** de la 0 la 255. Tipul **char** este considerat de compilator ca fiind distinct de celelalte două tipuri caracter, în expresii este promovat în mod implicit la un întreg de tip **int** din domeniul -128,...,127 sau, dacă programul este compilat cu opțiunea /J, la un întreg din domeniul 0,...,255.

Următorul program exemplifică utilizarea datelor de tip **unsigned char** ca numere naturale:

```
int main() {
    unsigned char x, y, z, t;
    x=250;
    y=4;
    z=x+y;
    t=z+7;
    cout<<"z="<<(int) z<<" t="<<(int) t<<endl;
    //                pe monitor:
    //                z=254 t=5
    //                Press any key to continue
    return 0;
}
```

Observăm că la execuția instrucțiunii `t=z+7;` are loc o depășire a domeniului de valori, depășire care este rezolvată după cum am aratat mai sus: prin pierderea celor mai semnificative cifre ale rezultatului. Mai precis, în scrierea hexazecimală avem

$$\text{FE}+\text{07}=\text{01 05.}$$

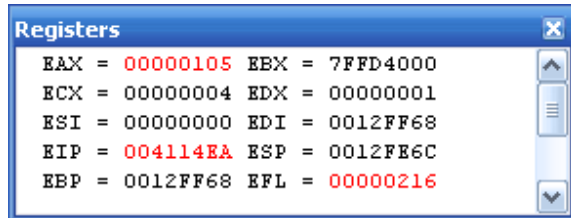
Deoarece rezultatul ocupă doi octeți, primul se pierde, și obținem

$$\text{FE}+\text{07}=\text{05}$$

adică

$$254+7=261=5 \pmod{256}.$$

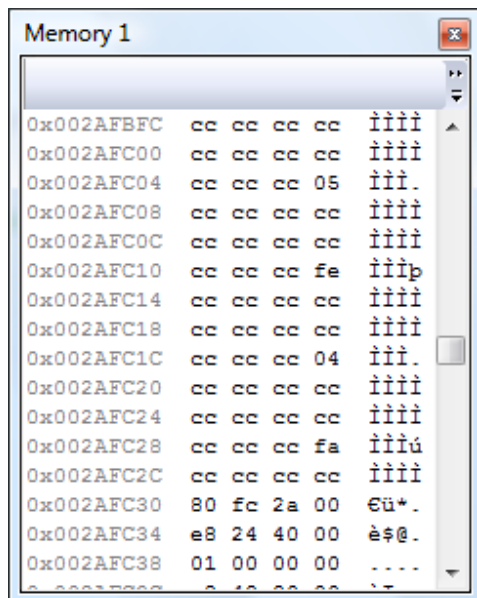
Iată rezultatul adunării așa cum apare în registrul EAX



și iată și instrucțiunea în limbaj de asamblare prin care se transferă în locația variabilei **t** numai ultimul octet din EAX, cel din subregistru AL:

```
004114EA mov byte ptr [t],al
```

Următoarea imagine a memoriei ne ilustrează alocarea datelor declarate în funcția **main**:



Variabila **x** are alocat octetul cu adresa 002AFC2B, **y** are adresa 002AFC1F, **z** are adresa 002AFC13 iar **t** are adresa 002AFC07. Observăm că octeții cu adresele 002AFC28, 29 și 2A (de exemplu) nu sunt folosiți, deoarece memoria este aliniată pe patru octeți și în consecință primul octet al unei locații de date trebuie să aibă adresa multiplu de patru. Rezultă că la declararea unei variabile de tip **char** ocupăm în memorie patru octeți din care folosim doar unul (pe ultimul, în cazul compilatorului MS Visual C++ 9.0) și deci, în cazul variabilelor simple, utilizarea tipului **char** în locul lui **int** pentru a face economie de memorie nu este justificată. În cazul tablourilor de caractere situația este alta, numai adresa primului caracter trebuie să fie multiplu de patru, următorii având adrese consecutive. Nu mai are loc aceasta risipă de memorie, totuși, întregul tablou fiind alocat într-o zonă de memorie care trebuie să aibă mărimea (în octeți) un număr multiplu de patru, pot să apară, după sfârșitul tabloului, unul, doi sau trei octeți nefolosiți.

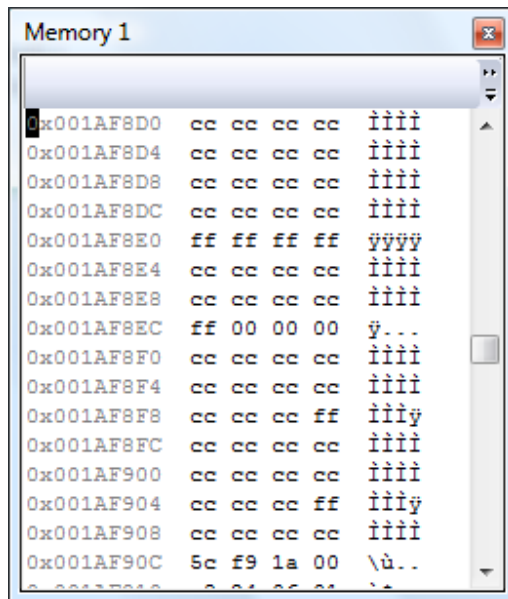
Analog datelor de tip **unsigned char**, și datele de tip **signed char** pot fi utilizate în calcule aritmetice, supunându-se și ele aritmeticii modulo 256, dar având acum drept reprezentanți ai claselor de resturi numere întregi din domeniul $-128, \dots, 127$. Exemplu:

```
int main(void) {
    signed char x, y, z, t;
    x=126;
    y=x+1;
    z=y+1;
    t=z+1;
    cout<<"y="<<(int) y<<" z="<<(int) z<<" t="<<(int) t<<endl;
    //          display:
    //          y=127 z=-128 t=-127
    //          Press any key to continue
    return 0;
}
```

Reprezentarea internă a tipului **signed char** (și implicit a lui **char**) este pe un octet în complement față de doi, mai precis octetii 00,...,7F sunt interpretați ca fiind 0,...,127 iar următorii octeți, 80, 81,...,FF, ca fiind -128, -127,...,-1. Compararea a două caractere se face după promovarea lor la tipul **int**, adică sunt comparate **valorile** celor două date și nu formatul intern de reprezentare, existând posibilitatea ca două date aritmetice de tip diferit să aibe același format intern dar valori diferite, ca în exemplul următor:

```
#include<iostream>
using namespace std;
int main() {
    unsigned char x=255;
    char y;
    y=x;
    if (x==y)    cout<<" DA " <<endl;
    else        cout<<" NU " <<endl;
    int ix,iy;
    ix=x;
    iy=y;
    cout<<"ix="<<ix<<endl;
    cout<<"iy="<<iy<<endl;
    //pe monitor:
    //    NU
    //ix=255
    //iy=-1
    //Press any key to continue . . .
    return 0;
}
```

Pe monitor apare NU deoarece $x=255$ iar $y=-1$, deși în memorie x și y arată la fel (dar nu și ix și iy):



Variabila **x** este la adresa 001AF907, are formatul intern FF și valoarea 255, fiind de tip **unsigned char**. Variabila **y** este la adresa 001AF8F7, are aceeași reprezentare internă, FF, dar valoarea sa este -1 (**y** fiind de tip **char**, valoarea sa se calculează exact ca în cazul **signed char**, adică: $255 \equiv -1 \pmod{256}$). Ca dovadă, variabila de tip **int** **ix** are reprezentarea internă **ff 00 00 00** (octeții 001AF8EC, 001AF8ED, 001AF8EE și 001AF8EF) iar **iy** are reprezentarea **ff ff ff ff**.

Este instructiv de studiat codul desasamblat. În fragmentul

```

    unsigned char x=255;
01131B3E  mov     byte ptr [x],0FFh
    char y;
    y=x;
01131B42  mov     al,byte ptr [x]
01131B45  mov     byte ptr [y],al
    if (x==y) cout<<" DA "<<endl;
01131B48  movzx   eax,byte ptr [x]
01131B4C  movsx   ecx,byte ptr [y]
01131B50  cmp     eax,ecx
01131B52  jne     main+61h (1131B81h)

```

observăm că atribuirea $y=x$ a fost efectuată prin subregistrul AL pe 8 biți, care a păstrat astfel formatul intern al datelor, în schimb comparația $x==y$ a fost efectuată după mutarea datelor în registrele pe 32 de biți **eax** și **ecx**. Observăm că **x** a fost mutat cu microinstrucțiunea **movzx**, care promovează un **unsigned char** la un **int**, iar **y** cu **movsx**, care promovează un **signed char** la un **int**. Aceste microinstrucțiuni apar și la atribuirile:

```

    int ix,iy;
    ix=x;
01131BAC  movzx   eax,byte ptr [x]
01131BB0  mov     dword ptr [ix],eax
    iy=y;
01131BB3  movsx   eax,byte ptr [y]
01131BB7  mov     dword ptr [iy],eax

```

Octeții *x* și *y* sunt identici dar nu au același tip, prin urmare, când sunt mutați în regiștrii microprocesorului pentru a obține valoarea lor numerică se folosesc instrucțiuni diferite și astfel, în acest caz, apar valori diferite.

În concluzie la cele de mai sus, o variabilă de tip **char** poate fi folosită în calcule la fel ca orice variabilă întregă (dar cu numai 256 de valori distincte posibile), ca indice al unui tablou, de exemplu, dar această utilizare a tipului **char** în locul lui **int** nu este justificată nici de economisirea spațiului de alocare, nici de claritatea programului, și poate duce la erori de execuție greu de depistat. Totuși, chiar și în cazul în care datele de tip **char** sunt utilizate conform scopului principal, ca fiind coduri ASCII ale unor caractere, apar situații în care utilizarea operațiilor aritmetice asupra lor este avantajoasă și nu trebuie evitată. De exemplu, dacă variabila *c* de tip **char** conține codul ASCII al unei litere mici, după executarea instrucțiunii

`c += 'A' - 'a' ;`

ea va conține codul literei mari corespunzătoare.

Reținem: tipul **char** este memorat pe un octet și în consecință calculele aritmetice cu variabile de tip caracter sunt calcule modulo 256 (deoarece $2^8=256$) și, analog, calculul aritmetic cu tipul **int** se desfășoară modulo 2^{32} (iar 2^{32} este un număr mare, dar nu prea mare – cam 4 miliarde).

3. Declarații de variabile

Numim *variabilă* o dată a cărei valoare poate varia în timpul execuției programului. Orice variabilă are alocată o anumită zonă de memorie în care i se păstrează valoarea curentă. Pentru a utiliza o anumită variabilă, de exemplu într-o expresie, trebuie să avem o modalitate de referire a ei. Cea mai simplă *referință* la o anumită variabilă constă în utilizarea unui *nume*, dat cu ajutorul unui *identificator*.

Un *identificator* este un cuvânt format numai cu următoarele caractere: literele *a, b, ... z* și *A, B, ... Z*, cifrele *0, ... 9* și liniuța de subliniere `_` (underscore). Primul caracter al unui identificator nu poate fi o cifră. Lungimea maximă a unui identificator depinde de implementare, pentru MS Visual Studio 2013 numai primele 2048 caractere sunt semnificative. Exemple corecte de identificatori definiți de utilizator: **alfa**, **primul**, **al2lea**, **a_pe_3**, **cel_mai_mic**, **ZERO**, **celMaiMare**, și exemple incorecte: **2la2**, **ampe&rsand**, **sizeof**. Ultimul exemplu este incorect deoarece coincide cu un *cuvânt cheie* al limbajului. Identificatorii **zet** și **Zet** sunt distincți deoarece compilatorul face distincție între literele mici și literele mari,

Atragem atenția că prin editarea fișierelor sursă în format Unicode se pot folosi și diacritice în scrierea identificatorilor, dar, deși MS Visual Studio îi acceptă, utilizarea unor astfel de identificatori nu este deloc recomandată.

Cuvintele cheie sunt identificatori rezervați, având o utilizare specială în scrierea programului și nu pot fi folosiți în alte scopuri. Cuvintele cheie pot aparține limbajului, de exemplu: **break**, **int**, **for**, **while**, sau pot fi rezervate pentru uzul compilatorului, de exemplu **__asm**, **__cdecl**, **__stdcall**, care sunt cuvinte cheie pentru MS Visual C++.

Setul cuvintelor cheie a fost extins cu fiecare fază a dezvoltării limbajului (K&R C, ANSI C, ISO C, C++) și este astăzi prea stufos pentru a fi învățat pe de rost înainte de a afla utilizarea fiecărui cuvânt cheie în parte. La definirea unui identificator, pentru a evita coliziunea cu un cuvânt cheie, este bine de reținut ca acestea din urmă sunt scrise numai cu litere mici. De asemenea, trebuie reținut faptul ca identificatorii scriși numai cu majuscule au o întrebuințare specială (pot fi constante predefinite utilizate de compilator).

În funcție de complexitatea datelor pe care le reprezintă, distingem următoarele categorii de variabile: *variabile simple* – formate dintr-o singură dată numerică; *tablouri* – formate din mai multe date de același tip memorate în locații adiacente; *structuri* – formate din date de tipuri diferite și, în final, *clase* – formate atât din date cât și din funcții capabile să prelucrez datele componente.

Numirea unei variabile simple se face în același timp cu declararea tipului său, printr-o *instrucțiune declarație* de forma următoare:

tip *identificator*;

când se declară o singură variabilă, sau de forma

tip *identificator* _1, *identificator* _2,..., *identificator* _n;

când se declară mai multe variabile de același tip.

De exemplu, următoarele două instrucțiuni declară variabilele **i**, **j** și **k** de tipul **int**, și variabilele **ch1** și **ch2** de tipul **char**.

```
int i, j, k;  
char ch1, ch2;
```

Declararea unei variabile poate fi făcută oriunde în program dar înainte de prima utilizare a variabilei declarate. Dacă o variabilă este declarată în corpul unei funcții, fără alte precizări suplimentare privind modul de memorare, ea poate fi utilizată numai în interiorul acelei funcții. O astfel de variabilă este numită *variabilă locală*. Variabilele locale sunt alocate în mod implicit pe stiva de execuție a programului și prin urmare ele apar și dispar odată cu execuția funcției în care sunt declarate. Apelurile funcțiilor se execută succesiv și din acest motiv variabilele locale nu au fost lăsate să fie *vizibile* din interiorul altor funcții. Dacă o variabilă este declarată în exteriorul oricărei funcții (instrucțiunea declarație este singura instrucțiune care poate să apară în exterior, toate celelalte instrucțiuni trebuie să aparțină corpului unei funcții), atunci ea devine *variabilă globală* și poate fi utilizată de toate funcțiile din program care urmează acelei declarații. Variabilele globale sunt alocate într-o zonă specială de memorie accesibilă permanent de către funcțiile programului.

Numim *bloc* o secvență de cod cuprinsă între acolade: { ... }, de exemplu corpul unei funcții este un bloc. Blocurile pot fi imbricate (cuprinse unele în altele), de exemplu: { bloc_1...{sub-bloc...}...}. O variabilă declarată în interiorul unui bloc este vizibilă numai în acel bloc și în toate sub-blocurile sale. Nu se admit *redeclarări* de variabile la același etaj al unui bloc. Putem *suprascrie* numele unei variabile declarate într-un bloc prin declararea unei noi variabile cu același nume în interiorul unui sub-bloc, ca în exemplul următor:

```
int main() {  
    int i;  
    i=1;  
    cout<<"i="<<i<<endl;
```

```

{
    int i;
    i=2;
    cout<<"i="<<i<<endl;
}
cout<<"i="<<i<<endl;
//    pe monitor:
//    i=1
//    i=2
//    i=1
//    Press any key to continue . . .
return 0;
}

```

Spunem că prima variabilă **i** (cea declarată la intrarea în **main**) a fost *redeclarată* (sau *suprascrisă*) în interiorul sub-blocului, dar de fapt în sub-bloc **i** este o nouă variabilă, vizibilă numai aici și care dispare la ieșirea din sub-bloc.

Variabilele declarate într-un program trebuie *inițializate* pentru a putea fi utilizate în mod corect. Inițializarea unei variabile simple constă în atribuirea unei valori inițiale, înainte de orice altă utilizare a respectivei variabile, și poate fi făcută odată cu declarea ei, astfel:

```
int a, b=5, c;
```

Mai sus au fost declarate variabilele **a**, **b** și **c** de tip **int**, iar **b** a fost inițializat cu valoarea 5. Utilizarea unei date ne-inițializate este o greșeală de programare.

Inițializarea unei variabile odată cu declararea se face, de obicei, prin atribuirea unei *constante* sau printr-o *expresie constantă* (expresie care poate fi evaluată de compilator în faza de compilare a programului). În C++ sunt admise și expresii care pot fi evaluate numai la rularea programului.

Celelalte categorii de date (tablouri, structuri, clase) au reguli speciale de inițializare.

Variabilele globale sunt pre-inițializate cu zerouri: următorul program

```

#include<iostream>
using namespace std;
int x;
int main() {
    cout<<"x="<<x<<endl;
    return 0;
}

```

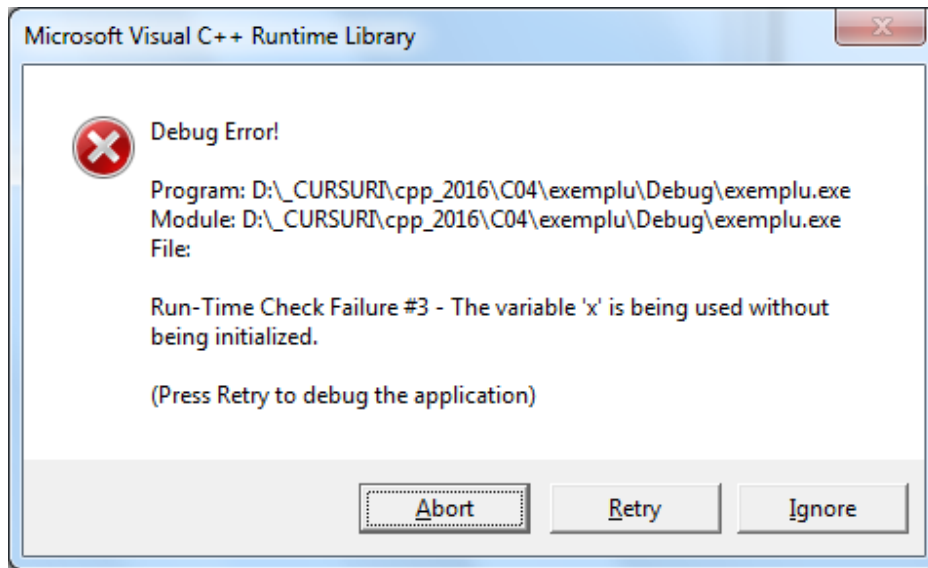
scoate pe monitor mesajul **x=0**, în timp ce următorul

```

#include<iostream>
using namespace std;
int main() {
    int x;
    cout<<"x="<<x<<endl;
    return 0;
}

```

după ce trece peste avertizarea debugger-ului



are ca rezultat $x = -858993460$, o valoare gunoi (*garbage value*) găsită în memorie la momentul alocării variabilei x

Dacă o declarație cu inițializare apare în corpul unei instrucțiuni de ciclare (**for** – de exemplu) datele sunt alocate o numai la prima execuție a corpului dar inițializarea lor se execută la fiecare ciclare. Instrucțiunea

```
for(int i=0;i<5;i++){
    int a=2;
    cout<<"adresa lui a="<<&a<<" a="<<a<<endl;
    a++;
}
```

alocă o singură dată variabila **a** dar o inițializează de 5 ori. Iată rezultatul rulării:

```
adresa lui a=0032F728 a=2
adresa lui a=0032F728 a=2
adresa lui a=0032F728 a=2
adresa lui a=0032F728 a=2
adresa lui a=0032F728 a=2
Press any key to continue . . .
```

Variabilele *tablou* pot fi uni- sau multi-dimensionale, în funcție de numărul de *indici* utilizați. Un tablou uni-dimensional (cu un singur indice) este numit *vector linie* sau numai *vector*. Analog, un tablou bi-dimensional este numit *matrice*. Toate elementele unui tablou au același tip, precizat la declararea tabloului. Următoarea instrucțiune declară un vector cu 5 componente, toate de tip **int**:

```
int vect[5];
```

Elementele sale sunt **vect[0]**, **vect[1]**, ..., **vect[4]**. Trebuie reținut că în limbajul C indicii unui tablou încep întotdeauna cu valoarea zero. Tabloul uni-dimensional **vect** are dimensiunea 5 (numărul de valori pe care îl poate lua singurul său indice). Un tablou tri-dimensional cu elemente de tip **double**, cu dimensiunile 5, 7 și 2, este declarat astfel:

```
double tab[5][7][2];
```

Referirea la un anumit element al tabloului **tab** se face în mod indexat, conform exemplului urmator:

```
tab[i][j][k]=13+i+3*j+i*k;
```

unde indicii **i**, **j** și **k** trebuie să fie întregi (sau expresii de tip întreg) cu valorile $i=0,\dots,4$; $j=0,\dots,6$ și $k=0,1$. Elementele unui tablou sunt alocate succesiv în memorie în ordinea obținută prin incrementarea ciclică a indicilor de la dreapta la stanga.

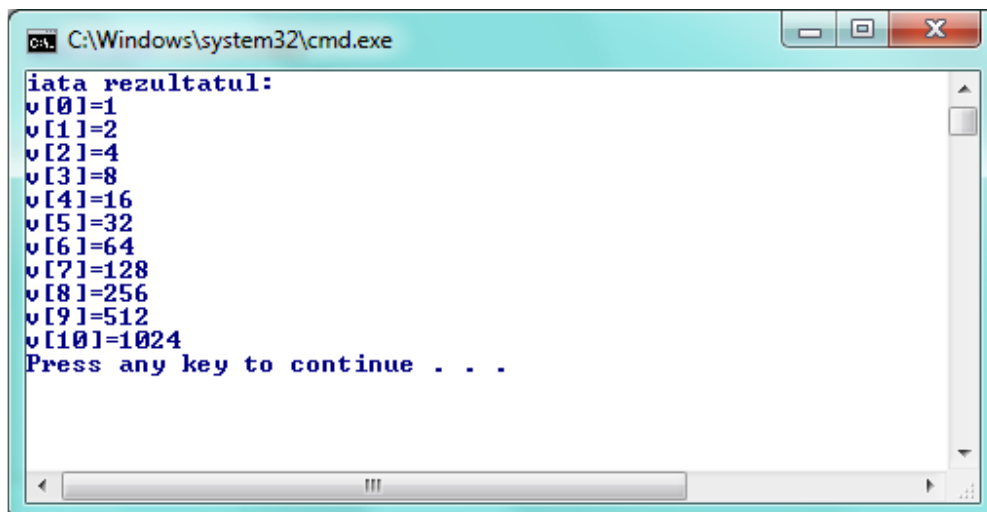
La declarare, dimensiunile unui tablou trebuie să fie exprimate prin constante sau, așa cum se vede mai jos, prin variabile declarate constante cu modificatorul **const** (specific C++). Această cerință se datorează faptului că alocarea memoriei are loc la compilare, înaintea rulării, așa că mărimea spațiului alocat unui tablou nu poate să depindă de rezultatul vreunui calcul din timpul execuției programului.

În programul următor calculăm și stocăm într-un vector primele zece puteri ale numărului 2:

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

const int dim=11;

int main() {
    int v[dim];
    int i;
    v[0]=1;
    for (i=1;i<dim;i++)
        v[i]=2*v[i-1];
    cout<<"iata rezultatul:"<<endl;
    for (i=0;i<dim;i++)
        cout<<"v["<<i<<"]="<<v[i]<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
iata rezultatul:
v[0]=1
v[1]=2
v[2]=4
v[3]=8
v[4]=16
v[5]=32
v[6]=64
v[7]=128
v[8]=256
v[9]=512
v[10]=1024
Press any key to continue . . .
```

Tablourile pot fi inițializate o dată cu declararea lor, cu sintaxa

tip_element nume_tablou[dimensiune]={ element_0, element_1, ...};

De exemplu, în instrucțiunea

```
int a[5]={0,10,20,30,40};
```

declaram că **a** este un vector cu cinci componente întregi pe care le inițializăm cu valorile $a[0]=0$, $a[1]=10$, ..., $a[4]=40$.

În lista de inițializare nu trebuie să fie mai multe elemente decât dimensiunea declarată a tabloului; dacă sunt mai puține compilatorul consideră că acestea sunt primele iar cele care lipsesc sunt egale cu zero, și acționează în consecință. Inițializarea

```
double b[5]={1.0, 1.1};
```

are ca efect $b[0]=1.0$, $b[1]=1.1$, $b[2]=0$, ..., $b[4]=0$.

La o declarație de tablou cu inițializare dimensiunea acestuia poate să nu fie declarată explicit, în acest caz compilatorul consideră că dimensiunea este dată de numărul de elemente din lista de inițializare. De exemplu, instrucțiunea

```
double b[]={1.0, 1.1};
```

declară tabloul **b** cu numai două elemente și le inițializează corespunzător listei.

Inițializatorii din lista de inițializare trebuie să fie constante (dacă tabloul este declarat ca variabilă globală) sau expresii care să poată fi evaluate în momentul alocării tabloului. În cazul tablourilor multi-dimensionale trebuie ținut cont că, în C, acestea sunt tablouri de tablouri: declarația

```
int c[2][3]={{0,1,2},{0,10,20}};
```

declară pe **c** ca fiind un tablou format din două linii, fiecare linie fiind la rândul ei un tablou de trei întregi; modul de inițializare decurge de aici, avem deci $c[0][0]=0$, $c[0][1]=1$, ..., $c[1][2]=20$. Limbajul permite să renunțăm la acoladele interioare, inițializarea de mai sus poate fi scrisă și așa:

```
int c[2][3]={0,1,2,0,10,20};
```

Atragem atenția că aceste reguli de inițializare ale tablourilor fac parte din sintaxa instrucțiunii de declarație și se aplică numai în acest cadru. Un tablou neinițializat la declarare poate fi ulterior încărcat numai element cu element: codul

```
int note[3];
note[0]=10;
note[1]=9;
note[2]=10;
```

este corect și are același efect ca declarația cu inițializare

```
int note[3]={10, 9,10};
```

în schimb, nici

```
int note[3];
note={10, 9,10};
```

și nici

```
int note[3];
note[3]={10, 9,10};
```

nu trec de compilare.