Cursul 3

Limbajul C în două ore

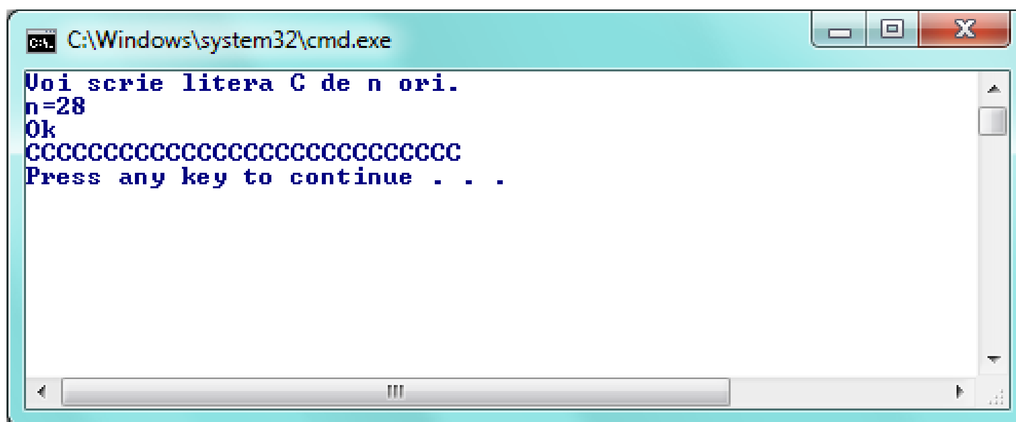
Vom face aici o prezentare generală a limbajului C (cu câteva elemente de C++) plecând de la rezolvarea următoarei probleme de programare:

Scrieţi un program care afişează pe monitor mesajul "**Voi scrie litera C de n ori.**", *după care scrie pe următorul rând mesajul* "**n=**" *şi aşteaptă ca utilizatorul să tasteze valoarea lui* **n**, *număr întreg.*

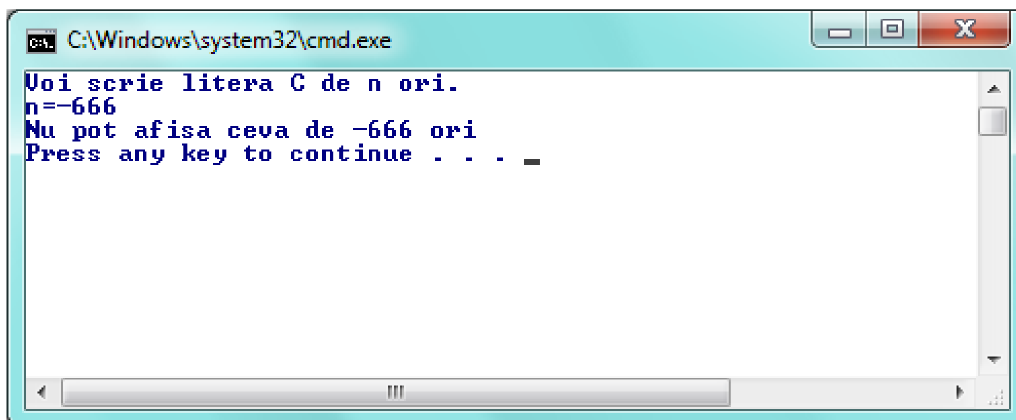
Dacă **n** *e strict pozitiv programul scrie* "**Ok**" *şi afişează de* **n** *ori litera* **C**, *altfel apare mesajul* "**Nu pot afisa ceva de ### ori**" *unde, pe monitor, în locul marcat cu* **###** *trebuie să apară valoarea lui* **n**.

Iată cum ar trebui să arate monitorul la finalul execuţiei în fiecare caz în parte:

cazul $n > 0$



şi cazul $n <= 0$



Incepem editarea fişierului sursă al programului, să îl numim **primul.cpp**, cu următoarea linie:

/* Primul program ilustrativ */

care conţine un *comentariu în stilul* C. Comentariile nu fac parte din program, ele au doar rolul de a da unele lămuriri celui ce citeşte textul sursă şi, mai ales în cazul programelor complexe, sunt foarte utile pentru înţelegerea diverselor secvenţe de cod.

Compilarea textului sursă se execută linie cu linie, de sus în jos, fiecare linie fiind citită de la stânga la dreapta. Compilatorul ignoră toate caracterele dintre perechile /* şi */ (inclusiv pe acestea), aşa că avem la dispoziţie un spaţiu în care putem scrie ce vrem. Un astfel de comentariu se poate întinde pe mai multe linii. Există şi *comentarii în stilul* C++, introduse odată cu extinderea la C++ a limbajului, acestea încep cu două slash-uri, //, şi se termină automat la capătul liniei, ele sunt utilizate pentru a da scurte indicaţii asupra conţinutului liniei de cod curente. Prezenţa comentariilor este facultativă, dar recomandabilă.

Următoarea linie este obligatorie:

 # include <iostream>

Această linie conţine o *directivă de preprocesare* şi nu face nici ea parte din program. După ce lansăm comanda "**Compile**" (Ctrl F7), compilatorul, înainte de a începe compilarea propriu-zisă a fişierului, lansează în execuţie preprocesorul – un program specializat în prelucrarea automată a textelor (prin substituţii) – care citeşte toate directivele de preprocesare şi le execută. In cazul directivei **include** linia directivei este înlocuită cu întreg conţinutul fişierului **iostream**, fişier care conţine tot ce este necesar programului pentru executarea operaţiilor de *citire/scriere* (sinonime: *intrare/ieşire*, *input/output*). Printre altele, prin acest mecanism ajung în fişierul nostru sursă definiţiile obiectelor **cout** şi **cin** (ca instanţe ale unor clase de fluxuri de date, *stream*-uri). *Identificatorul* **cout** desemnează consola de ieşire (*output console*), adică monitorul, iar **cin** desemnează consola de intrare (*input console*), adică tastatura. Fără această directivă de includere a bibliotecii **iostream** compilatorul ar da, la prima apariţie a cuvântului **cout,** mesajul de eroare **`'cout' : undeclared identifier`**. Alte biblioteci des utilizate: **<math.h>** pentru funcţii matematice (exponenţiale, logaritmi, funcţii trigonometrice, etc) şi **<string.h>** pentru operaţii cu şiruri de caractere (*string*-uri).

Subliniem de la bun început că textul sursă trebuie *editat* (scris) numai cu un set restrâns de caractere: literele a,b,c, ... , z, A,B,C, ... , Z, cifrele 0,1, ..., 9, şi simbolurile direct imprimabile de la tastatură: <, &, +, ş.a. Deşi MS Visual Studio acceptă acum folosirea caracterelor cu diacritice (ş, ţ, â, etc.) chiar şi în identificatori, nu recomandăm utilizarea acestei facilităţi.

Atenţie: compilatorul face distincţie între literele mari şi cele mici; de exemplu, dacă scriem în mod eronat

 # INCLUDE < iostream >

obţinem la compilare eroarea: **`invalid preprocessor command 'INCLUDE'.`**

De remacat şi faptul că, pentru o scriere mai lizibilă, se pot insera oricâte *blancuri* (cu tastele **space** sau **tab**) între *atomii lexicali* (*tokens*) ai directivei. E corect şi aşa

 # include < iostream >

şi aşa

 #   include   <   iostream   >

dar nu şi aşa

 # include < io   stream >

pentru că am spart un atom!

Această facilitate de spaţiere trebuie folosită pentru aşezarea în pagină, într-un mod cât mai clar, a textului sursă. In faza de preprocesare compilatorul şterge toate spaţiile care nu sunt neaparat necesare pentru păstrarea sensului directivelor şi al instrucţiunilor.

Directivele de preprocesare nu fac parte din limbajul C (nici din C++), ele sunt specifice fiecărui compilator în parte, existând totuşi o oarecare standardizare a lor. Ca regulă generală, ele încep cu simbolul "**#**" şi se termină "în aer", fără un simbol terminator (cum este, de exemplu, "**;**" pentru majoritatea instrucţiunilor C). Dintre directivele de preprocesare ale mediului de dezvoltare MS Visual C++ mai menţionăm doar directiva **#define** (vezi, de exemplu, https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives), des folosită pentru definirea *constantelor simbolice*, mai ales. De exemplu, în fişierul **iostream** există undeva directiva

#define EXIT_SUCCESS 0

care ne permite ca la sfârşitul funcţiei **main** să scriem instrucţiunea

return EXIT_SUCCESS;

în loc de

return 0;

Directiva **define** de mai sus are ca efect o substituţie de texte în faza de prepocesare: peste tot unde apare în textul sursă, şirul de caractere **EXIT_SUCCESS** este înlocuit cu şirul format în acest caz numai din caracterul **0**, astfel că în textul primit de compilator apare instrucţiunea obişnuită, **return 0;** pe care acesta o înţelege.

Urmează acum încă o linie de cod care va fi prezentă în toate programele noastre:

using namespace std;

şi care deseori este numită "directiva using" deşi ea este o instrucţiune C++. Denumirea poate crea confunzii, mai ales că exită chiar o directivă de preprocesare **#using**, ea este folosită pentru a sugera asemănarea dintre efectul acestei instrucţiuni şi o substituţie de texte efectuată la preprocesare: efectul instrucţiunii de mai sus este ca şi cum, de exemplu, identificatorul **cout** ar fi înlocuit cu **std::cout** peste tot unde apare în textul sursă. Ca dovadă, programul nostru va rula şi dacă în locul instrucţiunii de mai sus scriem următoarele trei directive **define**:

```
//using namespace std;
#define cout std::cout
#define cin std::cin
```

#define endl std::endl

Ca urmare a creşterii complexităţii programelor şi a utilizării unor biblioteci de funcţii şi de clase tot mai bogate, pentru a evita *coliziunile* tot mai frecvente dintre nume (utilizarea aceluiaşi identificator pentru obiecte diferite) în C++ a fost introdus mecanismul spaţiilor de nume: programatorul poate prefixa identificatorii utilizaţi, cu sintaxa

prefix::identificator

prefixul desemnând un anumit spaţiu de nume, **namespace**, un fel de nume de familie. Instrucţiunea noastră **using** declară că vom utiliza spaţiul de nume **std** (standard) şi în programul nostru este necesară pentru a simplifica referirea fluxurilor standard **cin** şi **cout**. Fără aceasta ar trebui să scriem **std::cin** în loc de **cin** şi **std::cout** în loc de **cout**.

Instrucţiunea "**using namespace std;**" este compusă din patru atomi lexicali: *cuvintele cheie* **using** şi **namespace** (aparţinând limbajului C++), identificatorul **std** (care denumeşte spaţiul de nume standard – cel mai des folosit) şi *terminatorul de instrucţiune* "**;**". Identificatorul **std** este definit undeva în codul inclus în programul nostru prin directiva "**#include<iostream>**", ca dovadă, dacă schimbăm ordinea acestor două linii obţinem la compilare eroarea: `'std' : a namespace with this name does not exist.`

Instrucţiunile limbajului C se termină, cele mai multe, cu simbolul "**;**". Instrucţiunile care nu se termină cu "**;**" sunt instrucţiunea compusă (blocul de instrucţiuni, { }) şi toate instrucţiunile care se pot termina cu un astfel de bloc de instrucţiuni (de exemplu **for**, **while**, etc). Subliniem că simbolul "**;**" nu este *separator* ci *terminator* de instrucţiuni: el face parte integrantă din instrucţiunea pe care o finalizează. De exemplu, pe linia

a=1; b=2; c=a+b;

am scris cu ajutorul a 14 caractere exact trei instrucţiuni, prima instrucţiune este scrisă cu 4 caractere, a doua tot cu 4, a treia cu 6 caractere, iar între instrucţiuni nu se află nimic. Regula este următoarea: în textul sursă, o instrucţiune începe imediat după ce se termină cea dinaintea ei. Reamintim că blancurile dintre tokeni sunt eliminate la preprocesare, aşa că pentru a face programul cât mai lizibil, instrucţiunile se scriu de regulă numai câte una pe linie şi cât mai aerisit.

Iată acum şi *codul sursă* al programului nostru:

```
/* Primul program ilustrativ */

#include<iostream>
using namespace std;

char lit = 'C';      //variabila globala

int citeste(){
       int n;        //variabila locala
       cout << "n=";
       cin >> n;
       return n;
}
```

```
void afiseaza(int nr){
	for (int i = 0; i < nr; i++){
		cout << lit;
	}
	return;
}

int main(){
	int k;
	cout << "Voi scrie litera " << lit << " de n ori." << endl;
	k = citeste();
	if (k <= 0)
		cout << "Nu pot afisa ceva de " << k << " ori!" << endl;
	else{
		cout << "Ok" << endl;
		afiseaza(k);
	}
	cout << endl;
	return 0;
}
```

Programul este compus dintr-o declaraţie de *variabilă* globală (variabila **lit**) şi trei definiţii de *funcţii* (funcţiile **citeste**, **afiseaza** şi **main**). In general, un program C este o colecţie de declaraţii de variabile globale şi de definiţii de funcţii (subprograme), dintre funcţii una (şi numai una) are numele **main**. Execuţia programului constă în prelucrarea valorilor unor date cu ajutorul acestor funcţii. In mod automat, prima care intră în execuţie este funcţia **main**, ea începe prelucrarea datelor conform instrucţiunilor sale componente, de regulă prin *apelarea* altor funcţii, succesiunea apelurilor fiind determinată de *fluxul de control* al programului.

Datele sunt numere sau coduri numerice, ele pot fi *constante* (valoarea lor este scrisă de programator direct în codul sursă) sau *variabile*, caz în care compilatorul trebuie să aloce spaţiul de memorie necesar păstrării valorilor curente ale respectivelor variabile. Orice dată trebuie să aibe un *tip* declarat în mod explicit de programator astfel încât compilatorul să ştie, în principal, cum să organizeze memoria alocată acelei date.

Cu *instrucţiunea declaraţie*:

char lit='C';

declarăm variabila **lit** ca fiind o dată de tip **char** şi o *iniţializăm* cu valoarea **'C'**. Cuvântul "**lit**" este *identificatorul* care dă *numele* variabilei, şi a fost ales de noi, spre deosebire de "**char**" care este un *cuvânt cheie* al limbajului (un *identificator rezervat*) şi care poate fi folosit numai pentru a desemna tipul de dată "caracter pe un octet".

Identificatorii pot fi formaţi numai din litere, cifre şi caracterul _ (*underscore*), cu restricţia că primul caracter nu poate fi o cifră. Lungimea maximă a unui identificator depinde de compilator (pentru MS Visual C++ vezi https://docs.microsoft.com/en-us/cpp/cpp/identifiers-cpp).

Variabila **lit** fiind declarată de tip **char**, la întâlnirea instrucţiunii de mai sus compilatorul rezervă un octet de memorie pentru variabila **lit** şi scrie în acest octet codul ASCII al literei 'C' (şi anume 43 hexa).

Tipurile uzuale pentru variabile simple sunt: **char** pentru caractere, **int** pentru numere întregi şi **double** pentru numere reale în dublă precizie (faţă de **float** – simplă precizie).

Variabila **lit** este *globală* deoarece a fost declarată în exteriorul oricărei funcţii. Valorile ei pot fi utilizate direct de către toate funcţiile care sunt definite după declararea ei, altfel spus, numele ei este *vizibil* din orice punct al fişierului sursă de după linia în care a fost declarată. Spre deosebire de ea, variabilele declarate în interiorul unei funcţii sunt *variabile locale*, ele pot fi referite numai în interiorul corpului funcţiei în care sunt declarate.

Declaraţiile pot fi făcute şi fără iniţializarea variabilei declarate, în felul următor:

```
char lit;
```

dar în acest caz trebuie să avem grijă ca, înainte de a o folosi, să îi *atribuim* variabilei **lit** o valoare iniţială, printr-o *instrucţiune expresie*, de exemplu,

```
lit = 'C';
```

Atenţie, spre deosebire de instrucţiunile declaraţii, care pot apare oriunde în textul sursă, instrucţiunile expresie pot să apară numai în interiorul corpului unei funcţii.

Să analizăm acum definiţia funcţiei **citeste**

```
int citeste(){
	int n;
	cout<<"n=";
	cin>>n;
	return n;
}
```

O funcţie este formată dintr-o *secvenţă de instrucţiuni* care prelucrează *datele de intrare* în funcţie (valorile actuale ale *argumentelor* funcţiei), împreună cu alte date proprii, şi care returnează către funcţia apelantă, o *dată de ieşire*, *rezultatul* funcţiei. Există şi funcţii fără date de intrare, precum şi funcţii care nu returnează nimic.

Din prima linie a definiţiei funcţiei **citeste** deducem că aceasta nu are date de intrare (acolo unde ar trebui să fie *lista parametrilor* nu este scris nimic, la fel de bine ar fi putut fi scris cuvântul cheie **void -** vacant) iar rezultatul ei este de tip **int** (adică va returna un număr întreg).

In general, definiţia unei funcţii are forma:

```
tip_returnat nume_funcţie (lista_parametrilor_funcţiei) {
	corpul_funcţiei
}
```

Dacă este nevoie, putem numai să declarăm funcţia (fără să o şi definim), scriind numai *prototipul funcţiei* (adică *antetul funcţiei* urmat de terminatorul de instrucţiune "**;**") astfel

```
tip_returnat nume_funcţie (lista_parametrilor_funcţiei);
```

dar tot va trebui să definim, undeva mai jos, înainte de sfârşitul fişierului sursă, funcţia.

Corpul funcţiei se scrie întotdeauna între două *acolade pereche*, {}, şi este format dintr-o secvenţă de instrucţiuni. Nu este permisă *imbricarea* funcţiilor, adică definirea unei funcţii în interiorul corpului alteia.

In corpul funcţiei **citeste** declarăm variabila **n**, scriem pe monitor mesajul **"n=",** citim de la tastatură valoarea variabilei **n** şi returnăm, ca rezultat al funcţiei, valoarea citită.

Pentru afişarea valorii unei variabile oarecare, **x** de exemplu, utilizăm *operatorul de scriere* , "<<", aplicat lui **cout** şi variabilei pe care dorim să o afişăm, în ordinea **cout << x.** Dacă dorim să afişăm un anumit mesaj, textul mesajului trebuie încadrat între ghilimele. De exemplu, instrucţiunea

cout << "aha";

va afişa textul **aha**. Intre secvenţele de cod

int x=123;
cout << x;

şi

int x=123;
cout << "x";

diferenţa este foarte mare, în primul caz pe monitor va fi afişat numărul 123, valoarea variabilei x, iar în al doilea caz va apare un text format doar dintr-un singur caracter: litera **x**.

Operatorul de scriere poate fi *concatenat*; pentru afişarea valorilor variabilelor **x**, **y** şi **z** putem folosi instrucţiunea

cout << x << y << z;

Pentru a distinge, pe monitor, valorile celor trei variabile e bine să intercalăm *blancuri*, spaţii de separare:

cout << x <<"   "<< y << "   "<< z;

sau chiar să fim şi mai expliciţi:

cout << "x=" << x <<" y= "<< y << " z= "<< z;

Pentru a trece la linie nouă (dar şi pentru a puncta încheierea transmisiei unui set de date) vom utiliza manipulatorul **endl**, astfel:

cout << "x=" << x <<" y= "<< y << " z= "<< z << endl;

Citirea de la tastatură se realizează cu *operatorul de citire*, " >> ", aplicat *stream*-ului **cin** şi variabilei citite, în ordinea **cin >> variabila**. Şi acest operator poate fi concatenat în cazul când vrem să citim mai multe valori, dar este mai sigur să scriem pentru fiecare citire câte o instrucţiune în parte sau chiar câte o pereche de instrucţiuni (prima pentru afişarea unui mesaj explicativ şi a doua pentru citirea propriu-zisă), aşa cum am întâlnit deja:

cout<<"n=";
cin>>n;

Execuţia unei funcţii se încheie când fluxul de control ajunge la execuţia unei instructiuni **return**, sau, dacă funcţia nu trebue să întoarcă nici un rezultat, la atingerea acoladei care încheie corpul funcţiei. Funcţia **citeste** se încheie cu instrucţiunea

return n;

care depune în registrul EAX al microprocesorului valoarea variabilei **n,** de unde este preluată de funcţia apelantă. In exemplul nostru, **citeste** este apelată o singură dată, în funcţia **main**, de instrucţiunea

k=citeste();

care atribuie variabilei **k** valoarea citită de funcţie de la tastatură. Atenţie, la apelarea unei funcţii fără argumente nu se foloseşte cuvântul cheie **void**.

Funcţia **afiseaza**

```
void afiseaza( int nr ){
        for( int i=0;  i<nr; i++ ){
                cout<<lit;
        }
        return;
}
```

are un singur argument, variabila **nr** de tip **int**, şi nu returnează nici un rezultat (spunem, prin abuz de limbaj, că rezultatul are tipul **void**). Ea nu calculează nimic, scrie numai pe monitor de exact **nr** ori caracterul dat de variabila **lit**. Corpul funcţiei este format din două instrucţiuni: o instrucţiune **for** şi o instrucţiune **return** fără valoare returnată (deorece funcţia nu trebuie să returneze nimic).

Instrucţiunea **for** este o *instrucţiune de ciclare*, de repetare a unei anumite acţiuni cât timp o anumită condiţie este îndeplinită. Acţiunea care se repetă poate fi oricât de complexă, ea este descrisă de *instrucţiunea corp* a **for**-ului, de regulă o instrucţiune compusă. Sintaxa (forma) instrucţiunii **for** este următoarea:

for (*iniţializare*; *expresie_test*; *expresie_de_actualizare*) **instructiune_corp**

Să urmărim ce se întâmplă în funcţia **afiseaza** când, de exemplu, **nr** are valoarea **7**. Execuţia **for**-ului începe prin executarea instrucţiunii de iniţializare "**int i=0**", care alocă variabila **i** locală **for**-ului (adică variabila **i** este vizibilă numai în interiorul acestui **for** şi îşi încetează existenţa odată acesta) şi îi atribuie valoarea iniţială 0, după care este evaluată expresia test "**i<nr**" care este adevărată (0<7). In acest caz se trece la execuţia instrucţiunii corp "**cout<<lit;**" şi apare un C pe ecran, după care, înainte de reluarea ciclării, este evaluată expresia de actualizare "**i++**", care îl incrementează pe **i** (îl măreşte cu o unitate). Acum **i** are valoarea 1 şi se reia testarea: **i<nr** ? da, apare încă un C, este incrementat **i**, şi aşa mai departe până când **i** atinge valoarea 7. Acum la evaluarea expresiei test obţinem valoarea logică *fals* şi execuţia **for**-ului este încheiată, prin urmare controlul execuţiei este preluat de instrucţiunea următoare, instrucţiunea "**return;**". Pe monitor au apărut astfel 7 litere C.

In exemplul nostru instrucţiunea corp a **for**-ului poate fi scrisă şi fără acolade, acestea sunt obligatorii numai când trebuie să executăm mai multe acţiuni în corpul **for**-ului, caz în care folosim pentru corp o *instrucţiune compusă*, grupând în interiorul unei perechi de acolade {} mai multe instrucţiuni simple (aşa cum avem de exemplu ramura **else** a instrucţiunii **if** din funcţia **main**). Începătorilor le recomandăm să folosească totdeauna acoladele, ca aici.

Să analizăm acum funcţia principală:

```
int main(){
        int k;
```

```
        cout << "Voi scrie " << lit << " de n ori." << endl;
        k = citeste();
        if ( k <= 0 )
                cout << "Nu pot afisa ceva de " << k<< " ori!" << endl;
        else{
                cout << "Ok" << endl;
                afiseaza(k);
        }
        cout << endl;
        return 0;
}
```

In general, funcţia principală poate avea unul dintre următoarele prototipuri:

```
int main( void );
int main( );
int main( int argc, char *argv[] );
```

care presupun că la sfârşitul execuţiei funcţia **main** returnează sistemului de operare un cod de eroare, valoarea zero însemnând "fără erori". Standardul C++ nu cere nici măcar existenţa explicită a instrucţiunii **return** finale, caz în care compilatorul trebuie să completeze codul ca şi cum ar exista **return 0** pe ultima linie. Compilatorul nostru, MS Visual C++, acceptă şi prototipul în stil C

```
void main( void );
```

care poate fi utilizat, desigur, dar cu reducerea *portabilităţii* codului sursă (unele compilatoare nu acceptă acest format, vezi https://en.wikipedia.org/wiki/Entry_point). De fapt, chiar dacă folosim această formă care nu trebuie să returneze nimic sistemului de operare, compilatorul se asigură că la sfârşitul apelului în registrul EAX se găseşte valoarea zero. De exemplu, funcţia

```
void main()
{
        cout << "Atentie la final avem:  xor  eax,eax " << endl;
}
```

are codul dezasamblat

```
void main()
{
00E75E30  push        ebp
00E75E31  mov         ebp,esp
00E75E33  sub         esp,0C0h
00E75E39  push        ebx
00E75E3A  push        esi
00E75E3B  push        edi
00E75E3C  lea         edi,[ebp-0C0h]
00E75E42  mov         ecx,30h
00E75E47  mov         eax,0CCCCCCCCh
00E75E4C  rep stos    dword ptr es:[edi]
```

```
       cout << "Atentie la final avem:  xor  eax,eax " << endl;
00E75E4E  mov         esi,esp
00E75E50  push        0E713D9h
00E75E55  push        0E7CF80h
00E75E5A  mov         eax,dword ptr ds:[00E8009Ch]
00E75E5F  push        eax
00E75E60  call        std::operator<<<std::char_traits<char> > (0E712A3h)
00E75E65  add         esp,8
00E75E68  mov         ecx,eax
00E75E6A  call        dword ptr ds:[0E80090h]
00E75E70  cmp         esi,esp
00E75E72  call        __RTC_CheckEsp (0E7132Ah)
}
00E75E77  xor         eax,eax
00E75E79  pop         edi
00E75E7A  pop         esi
00E75E7B  pop         ebx
00E75E7C  add         esp,0C0h
00E75E82  cmp         ebp,esp
00E75E84  call        __RTC_CheckEsp (0E7132Ah)
00E75E89  mov         esp,ebp
00E75E8B  pop         ebp
00E75E8C  ret
```

în care se observă că prima microinstrucţiune de după finalul funcţiei este

```
00E75E77  xor         eax,eax
```

care încarcă în registrul **EAX** valoarea zero printr-un **sau disjunctiv** pe biţi.

Forma cu argumente a funcţiei **main** este folosită pentru a transmite programului, în momentul lansării, anumite informaţii prin preluarea lor din linia de comandă.

In funcţia **main** din programul nostru declarăm o variabilă întreagă, **k**, o încărcăm cu valoarea returnată de funcţia **citeste** şi apoi o testăm dacă e mai mică sau egală cu zero. Dacă da, scriem pe monitor ceva, dacă nu, scriem altceva. Programul este gata, return zero (succes), stop.

Să remarcăm apariţia unei *instrucţiuni de selecţie*, instrucţiunea **if-else**. Ea are sintaxa

if (*expresie_test***)** *instructiunea_daca_da*

else *instructiunea_daca_nu*

Identificatorii **if** şi **else** sunt cuvinte cheie, prezenţa perechii de paranteze rotunde "( )"este obligatorie. Execuţia instrucţiunii este evidentă: se evaluează expresia test, dacă testul este îndeplinit se execută instrucţiunea *daca_da*, altfel se execută instrucţiunea *daca_nu*. In ambele cazuri, la terminarea execuţiei, controlul este preluat de instrucţiunea imediat urmatoare în textul sursă, cu excepţia cazului în care apare o *instructiune de salt* (cum ar fi **return**, de exemplu).

Există şi forma numai cu **if**, având sintaxa

if (*expresie_test***)** *instructiune_corp*

şi care, dacă testul este îndeplinit, inserează în fluxul de control al execuţiei instrucţiunea corp, altfel nu face nimic.

In mod obişnuit, expresiile test sunt *comparaţii* de forma **a<b**, **a<=b**, **a==b**, **a>=b** sau **a>b**. De reţinut că expresia **a==b** *testează* dacă **a** este egal cu **b**, în timp ce expresia **a=b** *atribuie* lui **a** valoarea lui **b** (operatorul "==" este *operatorul de comparaţie*, iar operatorul "=" este *operatorul de atribuire*). Deoarece în matematică semnul egal este folosit în ambele sensuri (şi comparaţie, şi atribuire), în limbajul C se produc adesea erori prin scrierea unui singur semn de

egalitate acolo unde ar trebui două semne. O astfel de eroare este greu de depistat, ea "nu iese la compilare" (compilatorul acceptă şi atribuirea **a=b** ca expresie test în **if** sau **for** – vom vedea mai târziu de ce), iar la rulare programul dă rezultate haotice. Pentru evitarea acestei situaţii unele compilatoare de C, când întâlnesc atribuiri în expresii condiţionale, avertizează utilizatorul asupra posibilei confuzii. Mai mult, unele limbaje de programare provenite din C, cum ar fi Java sau C#, au separat tipul logic de tipul aritmetic astfel încât această eroare de editare să fie depistată de compilator. MS Visual C++ nu dă nici un avertisment, aşa că trebuie să fim atenţi la comparaţii.

Să urmărim acum evoluţia mijloacelor de calcul prin prisma creşterii complexităţii: primile maşini mecanice de calcul efectuau doar calcule cu câte o singură operaţie (cum fac acum cele mai simple calculatoare de buzunar). Au apărut apoi calculatoare electronice capabile să efectueze mai multe operaţii la o singură comandă, să evalueze o întreagă expresie aritmetică (există şi astăzi calculatoare de buzunar care pot evalua formule). Următoarea etapă: calculatoare capabile să evalueze mai multe expresii la o singură comandă, prin rularea unui program format dintr-o secvenţă de instrucţiuni executate într-o ordine dictată de rezultatele expresiilor evaluate. Iniţial calculatoarele programabile dispuneau de puţină memorie internă, de ordinul kiloocteţilor, aşa că programele trebuiau să fie scurte, cu câteva instrucţiuni; pentru prelucrări complexe ele au fost organizate ca subprograme (funcţii, proceduri, subrutine, etc.) şi încărcate în memorie doar atunci când erau apelate. După cum am văzut deja, un program C este o colecţie de astfel de subprograme.

Evoluţia a continuat: pentru utilizarea mai multor programe C în acelaşi timp, acestora li s-au scos funcţiile **main** şi au fost transformate în *clase*, variabilele globale devenind *variabile membru* iar funcţiile rămase devenind *funcţii membru* ale claselor. Incărcarea lor în memorie a fost transformată în instanţierea claselor, iar lansările în execuţie au fost înlocuite cu apeluri către funcţiile membru, apeluri făcute prin intermediul instanţelor (obiectelor) clasei. Pe scurt, în C++ a fost introdus tipul **class,** programatorul având posibilitatea să-şi definească propriile clase sau să folosească unele deja existente; prin definirea unei clase se defineşte un nou tip de dată – având complexitatea unui vechi program C –, pentru utilizarea unei clase trebuie să declarăm şi să iniţializăm o variabilă de tipul dat de acea clasă, adică să *instanţiem* un *obiect* al clasei.

De exemplu, în fişierul **iostream** din directorul **\include** al compilatorului MS Visual C++ găsim următoarea declaraţie:

extern ostream &cout;

Această instrucţiune declară **cout** ca fiind o referinţă (iniţializată într-un fişier extern) către un obiect din clasa **ostream,** care este o clasă specializată în operaţii de ieşire (*output stream*). Tot în fişierul **iostream** găsim directiva **#include<istream>**, iar în fişierul **istream** găsim **#include<ostream>**. Noi am inclus în fisierul nostru sursă fişierul **iostream** şi astfel, prin includeri succesive, definiţia clasei **ostream** şi declaraţia lui **cout** ajung în fişierul sursă al programului nostru, transformându-l într-un program C++, deşi în liniile de cod scrise de noi nu este definită nici o clasă.

In final, deoarece programele noastre C sunt de fapt programe C++, trebuie să precizăm următoarele: limbajul C++ este o extensie a limbajului C realizată prin adăugarea tipului de dată **class**, un program C++ este format din declaraţii de variabile globale şi din definiţii de clase şi de funcţii care manipulează obiecte din aceste clase, dintre funcţii una se numeşte **main** şi cu ea începe execuţia programului. Această extindere a limbajului a fost iniţiată în 1979 de către Bjarne Stroustrup, cercetător la AT&T Bell Laboratories (tot acolo unde a fost creat şi limbajul C), care iniţial a numit noul limbaj "C with Classes" şi apoi, în 1983, i-a dat denumirea definitivă de C++. Manualul fundamental a fost scris de Bjarne Stroustrup în 1985, *The C++ Programming Language*. Pentru mai multe amănunte, vezi pagina personală a autorului, http://www.stroustrup.com/.

Pentru a ilustra cele spuse mai sus despre noţiunea de clasă, prezentăm în final o rezolvare în stil C++ a problemei noastre, obţinută prin schimbări minimale, prin transformarea programului C prezentat mai sus într-o clasă C++, clasa **scriitor**:

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

class scriitor{
public:
	char lit;
	int citeste(){
		int n;
		cout << "n=";
		cin >> n;
		return n;
	}
	void afiseaza(int nr){
		for (int i = 0; i < nr; i++)
			cout << lit;
		return;
	}
	void lucreaza(){
		int k;
		cout << "Voi scrie litera " << lit << " de n ori." << endl;
		k = citeste();
		if (k <= 0)
			cout << "Nu pot afisa ceva de " << k << " ori!" << endl;
		else{
			cout << "Ok" << endl;
			afiseaza(k);
		}
		cout << endl;
	}
};

int main(){
	scriitor ionescu;
	ionescu.lit = 'C';
	ionescu.lucreaza();
	return 0;
}
```