Viorel PĂUN

Programarea în limbajul C/ C++

Cuprins

INTRODUCERE

Crearea limbajului C a început prin anii ‘ 70 de către Dennis Ritchie (născut în 1941 în SUA) pe baza unor preocupări anterioare ale altor specialişti.

În anul 1978, Brian Kernigan(născut în 1942 în Canada) şi Dennis Ritchie de la Bell Laboratories, New Jersey-SUA, publică “The C Programming Language” în care este descrisă o variantă clasică a limbajului C, folosind un sistem de operare UNIX.

În anul 1983, un comitet ANSI (American National Standard Institute) a început redactarea limbajului C standard, finalizat în anul 1990.

Firma Borland a elaborat diverse versiuni Turbo C, Turbo C++ etc.

Limbajul C a avut un succes deosebit în rândul programatorilor.

Numărul mare de aplicaţii realizate în limbajele C şi C++ confirmă faptul că la ora actuală aceste limbaje de programare sunt foarte populare.

Acest lucru se vede şi din statistica popularităţii limbajelor de programare furnizată de TIOBE Software, specializată în evaluarea şi urmărirea calităţii software-ului. Indicele de popularitate este actualizat o dată pe lună.

Position May 2011	Position May 2010	Delta in Position	Programming Language	Ratings May 2011	Delta May 2010
1	2	↑	Java	18.160%	+0.20%
2	1	↓	C	16.170%	-2.02%
3	3	=	C++	9.146%	-1.23%
4	6	↑↑	C#	7.539%	+2.76%
5	4	↓	PHP	6.508%	-2.57%
6	10	↑↑↑↑	Objective-C	5.010%	+2.65%
7	7	=	Python	4.583%	+0.49%
8	5	↓↓↓	(Visual) Basic	4.496%	-1.16%
9	8	↓	Perl	2.231%	-1.05%
10	11	↑	Ruby	1.421%	-0.67%

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Limbajul C s-a născut ca o alternativă pentru limbajele de asamblare, fiind utilizat pentru scrierea unor sisteme de operare, compilatoare, editoare de texte etc, dar, în acelaşi timp este un limbaj de nivel înalt, folosit pentru aplicaţii din cele mai diverse domenii.

Limbajul C este conceput pe principiul programării structurate.

Iniţial, limbjul C a fost utilizat pentru a substitui limbajul de asamblare în aplicaţii mai ample, de la sisteme de operare şi compilatoare până la cele mai diverse aplicaţii.

De exemplu, sistemul de operare UNIX este scris aproape în totalitate în C.

Caracterul sau universal constă în îmbinarea elementelor de limbaj de nivel înalt, cu o serie de operaţii tipice limbajelor de asamblare.

Setul de instrucţiuni şi de date este similar cu cel din limbajul PASCAL.

Prin comparaţie, setul de instrucţiuni C este mai cuprinzător decât cel al limbajului Pascal.

Posibilităţile de lucru cu adrese de date şi de funcţii (pointeri), inclusiv efectuarea de operaţii aritmetice şi comparaţii, posibilitatea de a efectua operaţii la nivel de bit, apropie limbajul C de limbajele de asamblare. Acestea oferă posibilitatea utilizatorului să lucreze cu elementele hardware ale calculatorului.

În condiţiile perfecţionării tehnicii de programare orientată pe obiecte (Object Oriented Programming), pe baza limbajului C, la sfârşitul anilor ‘80 a fost creat C++, ca o dezvoltare a acestuia. Bjarne Stroustrup (născut la data de 11 iunie 1950 în Aarhus, Danemarca) este cel care a creat limbajul de programare C++, a scris primele sale definiţii, a realizat prima implementare şi a fost responsabil pentru procesarea propunerilor de extindere a limbajului C++ în cadrul comitetului de standardizare. Primul plus simbolizează completările aduse limbajului C, dându-i mai multă flexibilitate, eficienţă şi rigurozitate. Al doilea plus se referă la acea dezvoltare care pune la dispoziţia programatorului tehnica programării orientată pe obiecte.

Programarea orientată pe obiecte se bazează pe noţiunea de obiect, care reprezintă un ansamblu format dintr-o structură de date şi metode (funcţii) de operare cu aceste date.

Capitolul I. BAZELE PROGRAMĂRII ÎN LIMBAJUL C/C + +

I.1. Prezentarea limbajului C: vocabular, tipuri de date, constante, variabile, operatori şi expresii

Un *program* este un text ce specifică acţiuni care vor fi executate de un procesor. Scrierea unui program se face într-un limbaj de programare.

Un *limbaj de programare* are vocabular şi reguli de sintaxă.

Vocabularul este format din unităţi lexicale: cele mai simple elemente cu semnificaţie lingvistică.

Sintaxa: ansamblu de reguli pe baza cărora se combină elementele vocabularului (unităţile lexicale) pentru a obţine fraze corecte (instrucţiuni, secvenţe de instrucţiuni, declarare de variabile etc).

Elementele vocabularului sunt alcătuite din caractere.

Orice caracter este reprezentat în calculator în codul ASCII (American Standard Code for Information Interchange), printr-un număr natural unic, cuprins între 0 şi 255 .

Caracterele sunt simboluri grupate în litere, cifre şi semne speciale, astfel:

- literele mari ale alfabetului englez cu codurile 65,...,90: A, B, ..., X, Y, Z
- literele mici ale alfabetului englez cu codurile 97,...,122: a, b, ..., x, y, z
- cifrele bazei zece, care au codurile in 48,...,57: 0, 1, 2, ... ,9
- liniuţa de subliniere _
- semne de punctuaţie şi semne speciale : , . : ? ‘ ( ) [ ] { } < > ! | \ / ~ # & ^ + - * %

Unităţile lexicale ale limbajului C/C++ sunt: identificatori, cuvinte cheie, constante, şiruri, operatori şi separatori. La scrierea lor se folosesc reguli precise date de sintaxa limbajului prin utilizarea setului de caractere al codului ASCII.

I.1.1. Identificatori şi cuvinte cheie

Un *identificator* reprezintă o succesiune de litere (litera mică este tratată ca distinctă de litera mare), cifre, liniuţe de subliniere, primul caracter din secvenţă fiind obligatoriu o literă sau o liniuţă de subliniere.

Identificatorii sunt *nume simbolice* date de utilizator constantelor, variabilelor, tipurilor de date, funcţiilor etc, pentru a descrie datele de prelucrat (de exemplu nume de variabile) şi procesele de prelucrare (de exemplu nume de funcţii). În general numai primele 32 de caractere se consideră semnificative în C.

Ca identificatori se preferă folosirea unor nume sau simboluri care să sugereze semnificaţia mărimilor pe care le desemnează, contribuind la creşterea clarităţii programului.

Cuvintele cheie (keywords) sunt cuvinte rezervate pentru limbaj în sine, au înţeles predefinit şi nu pot avea altă utilizare. Aceste cuvinte se scriu cu litere mici.

ANSI (American National Standards Institute) C are 32 de cuvinte cheie:

```
auto      const     double    float     int       short
struct    unsigned  break     continue  else      for
long      signed    switch    void      case      default
enum      goto      register  sizeof    typedef   volatile
char      do        extern    if        return    static
union     while
```

I.1.2. Comentariu

În redactarea programelor se folosesc o serie de texte care dau explicaţii cu privire la program, la părţile sale la variabilele utilizate etc. Aceste texte explicative se adresează utilizatorilor; se numesc comentarii şi sunt ignorate de compilator.

În limbajul C un comentariu începe prin **`/*`** şi se termină prin **`*/`**.

În C++ un comentariu care încape pe un rând poate fi scris după **`//`**.

I.1.3. Tipuri de date, constante, variabile

Tipul unei date determină:

- spaţiul de memorie ocupat;
- modul de reprezentare internă;
- domeniul de valori;
- timpul de viaţă asociat datei;
- operatori utilizaţi şi restricţii în folosirea acestora.

Limbajul C lucrează cu valori care pot fi stocate în constante sau variabile.

Constantele stochează valori nemodificabile pe parcursul execuţiei programului.

Variabilele sunt mărimi care îşi modifică valoarea în timpul execuţiei programului.

Tipurile de date utilizate de limbajul C se clasifică astfel:

- *tipuri fundamentale*
  - caracter
  - întregi
  - reale
  - tip void
- *tipuri derivate*
  - tablouri
  - pointeri
  - structuri
  - uniuni
  - enumerări

- tip definit de utilizator

Tipurile fundamentale se mai numesc scalare, predefinite, simple sau de bază.

I.1.4. Tipuri de date standard

Datele reprezintă informaţii care fac obiectul prelucrărilor. Fiecare dată este memorată într-un anumit format. Pe de altă parte, interpretarea valorilor memorate se face diferit, în funcţie de semnificaţia datelor respective. Prin urmare, este important atât modul de memorare a datelor (formatul fizic de reprezentare) prin care se stabileşte domeniul valorilor datelor, cât şi semnificaţia lor prin care se stabilesc operaţiile care se pot efectua cu aceste date. Aceste caracteristici ale unei date sunt precizate prin tipul său.

Tipurile de date standard (predefinite) ale limbajului C sunt:

Specificator	Abreviaţia	Lungime în biţi	Domeniu de valori
signed char	char	8	Caracter reprezentat prin cod ASCII sau întreg binar din intervalul –128 ... 127.
unsigned char		8	Caracter reprezentat prin cod ASCII sau întreg binar fără semn din intervalul 0 ... 255.
signed int	int	dependentă de calculator 16 sau 32	Întreg binar reprezentat în cod complementar faţă de 2
short int	short	16	Întreg binar reprezentat în cod complementar faţă de 2, din intervalul -32768 ... 32767.
long int	long	32	Întreg binar reprezentat în cod complementar faţă de 2 din intervalul -2147483648 ... 2147483647
unsigned int	unsigned	dependentă de calculator 16 sau 32	Întreg binar fără semn
unsigned short int	unsigned short	16	Întreg binar fără semn din intervalul 0 ... 65535.
unsigned long int	unsigned long	32	Întreg binar fără semn din intervalul 0 ... 4294967295
float		32	Număr reprezentat în virgulă mobilă 1bit pentru semn, 7b exponent, 24b mantisa, precizie 7 zecimale. Domeniu [3.4E-38, 3.4E38]
long float	double	64	Număr reprezentat în virgulă mobilă 1 bit ptr. semn, 11b exponent, 52b mantisa, precizie 15 zecimale.

			Domeniu [1.7E-308, 1.7E308]
long double		80	Număr reprezentat în virgulă mobilă precizie 19 zecimale. Domeniu [3.4E-4932, 1.1E4932]

Pe lângă aceste tipuri de date, limbajul C mai dispune de tipul *pointer* şi tipul *void*. Tipul *void* indică absenţa oricărei valori.

Pointerii se utilizează pentru a face referire la date cunoscute prin adresele lor. Un pointer este o variabilă care are ca valori adrese.

Tipul pointer are formatul:

```
<tip>* <nume>;
```

ceea ce înseamnă că *<nume>* este un pointer către o zonă de memorie ce conţine o dată de tipul *<tip>*.

I.1.5. Constante

O constantă este o valoare fixă care apare literalmente în codul sursă al unui program. Tipul și valoarea constantei sunt determinate de modul în care constanta este scrisă. Constantele pot fi de mai multe tipuri: întreg, flotant(real), caracter, șir de caractere. Aceste constante sunt folosite, de exemplu, pentru a iniţializarea variabilelor.

Constante întregi

O constantă întreagă este un număr întreg reprezentat în cod complementar faţă de 2 pe 16 biţi sau pe 32 biţi dacă nu încape pe 16 biţi. Exemple: 7, -3, 0.

În cazul în care dorim ca o constantă întreagă din intervalul -32768 ... +32767 să fie reprezentată pe 32 biţi (implicit astfel de constante se reprezintă pe 16 biţi), vom termina constanta respectivă prin L sau l, adică îi impunem tipul long. Exemplu `-10L`.

Dacă, la o constantă întreagă, adăugăm sufixul U sau u, atunci forţăm tipul constantei la unsigned int sau unsigned long. Dacă adăugăm sufixul UL sau ul sau Ul sau uL constanta va fi de tipul unsigned long.

O constantă întreagă, precedată de un zero nesemnificativ se consideră scrisă în sistemul de numeraţie cu baza 8.

O constantă întreagă care începe cu `0X` sau `0x` se consideră scrisă în sistemul de numeraţie cu baza 16 (cifrele hexazecimale sunt `0...9`, `a...f` sau `A...F`). În rest se consideră că baza de numeraţie este 10.

Exemple:

Constanta	Tip	Constanta	Tip
`1234`	`int`	`123456789L`	`long`
`02322`	`int` /* octal */	`1234U`	`unsigned int`
`0x4D2`	`Int` /* hexazecimal */	`123456789UL`	`unsigned long int`

Constante flotante

Atunci când încercăm să reprezentăm în memoria calculatorului un număr real, căutăm de fapt cel mai apropiat număr real reprezentabil în calculator şi aproximăm numărul iniţial cu acesta din urmă. Ca rezultat, putem efectua calcule complexe cu o precizie rezonabilă.

Constantele reale sunt reprezentate în virgulă mobilă prin notaţia clasică cu mantisă şi exponent.

Sintaxa: ±partea întreagă . partea fracţionară {E|e} ±exponentul

Pot lipsi fie partea întreagă, fie partea fracţionară, dar nu ambele.

Pot lipsi punctul zecimal cu partea fracţionară sau litera E cu exponentul dar nu ambele. Semnul + este opţional pentru numerele nenegative.

Exemple:

```
3.1415921    -12.              .34       -.125
4.3E20                        /* pentru numărul 4.3*10^20 */
-.2e+15                       /* pentru numărul -0.2*10^15 */
2e-5                          /* pentru numărul 2*10^-5 */
```

În mod implicit, o constantă reală este reprezentată intern în format double. Tipul constantei poate fi influenţat prin adăugarea unui sufix de f (sau F) sau l(sau L). Sufixul f (saui F) forţează constanta la tipul float, sufixul l (sau L) forţează constanta la tipul long duble.

Constanta	tip
12.34	double
12.3e-4	double
12.34F	float
12.34L	long double

Constante caracter

O constantă caracter reprezintă un caracter şi are ca valoare codul ASCII al caracterului respectiv. O constantă caracter grafic se poate scrie incluzând caracterul respectiv între caractere apostrof.

Exemple:

- literele mari au codurile ASCII în intervalul [65,90]
  'A' —> 65, ... , 'Z' —> 90.
- literele mici au codurile ASCII în intervalul [97,122]
  'a' —> 97, ... ,'z' —> 122
- cifrele au codurile ASCII în intervalul [48,57]
  '0' —> 48, ... ,'9' —> 57
- constanta '*' are valoarea 77

Caracterele negrafice cu excepţia caracterului DEL (care are codul 127), au coduri ASCII mai mici decât 32. O parte dintre aceste caractere formează categoria caracterelor de control şi au notaţii speciale de tipul `\caracter`.

De exemplu codul ASCII de valoare zero defineşte caracterul NULL. Acesta este un caracter impropriu şi spre deosebire de alte caractere el nu poate fi generat de la tastatură şi nu are efect nici la ieşire. În C este folosit ca terminator pentru şiruri de caractere şi are notaţia \0.

Setul de caractere de control:

Valoare cod ASCII	Reprezentare	Rol
0	\0	Caracterul NULL (zero binar)
7	\a	Alarmă (bell)
8	\b	Spaţiu înapoi (backspace); BS
9	\t	Tabulator orizontal; TAB
10	\n	Salt la linie nouă (new line)
11	\v	Tabulator vertical
12	\f	Salt de pagină la imprimantă (formfeed); FF
13	\r	Deplasarea cursorului în coloana 1 pe aceeaşi linie; CR

O constantă caracter cu notaţie specială, se va scrie incluzând notaţia între caractere apostrof. Exemple: '\n', '\t', '\r'

Constanta *apostrof* se reprezintă prin `'\''`.

Constanta *backslash* se reprezintă prin `'\\'`.

Construcţia `'\ddd'`, unde `d` este o cifră octală, reprezintă caracterul al cărui cod ASCII are valoarea egală cu numărul octal ddd. În particular caracterul impropriu NULL se poate reprezenta prin constanta caracter `'\0'`.

Caracterul DEL al cărui cod ASCII are valoarea 127 se reprezintă prin `'\177'`. Caracterul spaţiu al cărui cod ASCII are valoarea 32 se poate reprezenta prin `' '` sau `'\40'`.

Construcţia `'\xdd'`, unde d reprezintă o cifră hexazecimală reprezintă caracterul al cărui cod ASCII are valoarea egală cu numărul zecimal dd.

Exemplu: `'\x20'` reprezintă caracterul spaţiu.

Dacă `\` este urmat de un alt caracter decât cele arătate, atunci `\` este ignorat de compilator.

Caracterele *spaţiu* (cod ASCII 32), *tab* (cod ASCII 9) şi *linie nouă* (cod ASCII 10) formează categoria de separatori "*spaţii albe*". În afară de locurile unde sunt necesare spaţiile albe pentru separarea identificatorilor, a cuvintelor cheie etc. aceste spaţii albe sunt ignorate de compilator şi pot fi folosite oriunde în program.

Constante şir de caractere

O constantă *şir de caractere* este o succesiune de zero sau mai multe caractere delimitate prin ghilimele. Ghilimelele nu fac parte din şirul de caractere. Dacă dorim să folosim caractere negrafice în compunerea unui şir de caractere, atunci putem folosi convenţia de utilizare a caracterului `\`. Dacă dorim să reprezentăm chiar caracterul ghilimele, atunci vom scrie `\"`, de asemenea pentru backslash scriem `\\`.

Exemple:

```
"123"
"1\"2"              -reprezintă succesiunea 1"2
"a\\b"              -reprezintă succesiunea a\b
"c:\\tc\\bgi"       -reprezintă succesiunea c:\tc\bgi
```

Un şir de caractere poate fi continuat de pe un rând pe altul, dacă înainte de a acţiona tasta <enter> se va tasta \.

Constanta şir de caractere se reprezintă printr-o succesiune de octeţi în care se păstrează codurile ASCII ale caracterelor şirului, iar ultimul octet conţine totdeauna caracterul NULL pentru a marca sfârşitul şirului. De aici rezultă că, de exemplu, `'A'` şi `"A"` sunt construcţii diferite. Prima reprezintă o constantă caracter care se păstrează pe un singur octet în memorie. A doua, reprezintă un şir de caractere şi se păstrează pe doi octeţi, primul octet conţine valoarea codului ASCII al lui A, iar cel de-al doilea conţine caracterul NULL, adică valoarea 0.

I.1.6. Variabile

Prin variabilă înţelegem o zonă temporară de stocare a datelor a cărei valoare se poate schimba în timpul execuţiei programului. Unei variabile i se asociază un nume prin intermediul căruia putem avea acces la valoarea ei şi un tip de date care stabileşte valorile pe care le poate lua variabila. Corespondenţa între numele şi tipul unei variabile se realizează printr-o construcţie specială numită *declaraţie*. Toate variabilele utilizate într-un program trebuie declarate înaintea utilizării lor.

O declaraţie de variabilă are următoarea sintaxă:

< tip > < lista de variabile >

unde lista conţine unul sau mai multe nume de variabile despărţite prin virgule.

Exemple:

```
int i,j,X;
unsigned long k;
float a,b;
char c;
```

Sunt situaţii în care variabilele trebuie să fie grupate din punct de vedere logic. Tablourile reprezintă grupuri unidimensionale sau multidimensionale de variabile de acelaşi tip. Declaraţia de tablou conţine tipul comun al elementelor sale, numele tabloului şi numărul de elemente pentru fiecare dimensiune incluse între paranteze drepte.

```
<tip>  <lista de elemente>;
```

Elementele se separă prin virgule.

Un element din lista de tip tablou are formatul:

```
nume[dim1][dim2]...[dimn]
```

unde $dim_1, dim_2, ..., dim_n$ sunt expresii constante care au valori întregi.

Exemple:

```
int v[10];
float a[100][3];
```

La elementele unui tablou ne referim prin variabile cu indici. O astfel de variabilă se compune din numele tabloului urmat de unul sau mai mulţi indici, fiecare indice fiind inclus între paranteze drepte. Indicii au limita inferioară zero.

Exemple: Tablourile `v` şi `a` declarate mai sus se compun din variabilele

```
v[0], v[1], ... , v[9]
```

respectiv,

```
a[0][0],   a[0][1],   a[0][2],
a[1][0],   a[1][1],   a[1][2],
                ...
a[99][0]   a[99][1],   a[99][2].
```

Tablourile unidimensionale de tip caracter se utilizează pentru a păstra şiruri de caractere. Exemplu:

```
char tab[4];
```

tab poate păstra un şir de maxim 3 caractere, al patrulea octet fiind necesar pentru caracterul NULL - marcatorul sfârşituli şirului.

Numele unui tablou are ca valoare adresa primului său element.

I.1.7. Operatori şi expresii

Operatorii sunt simboluri care specifică operaţiile ce se aplică unor variabile sau constante numite operanzi.

O expresie este o construcţie aritmetică sau algebrică care defineşte un calcul prin aplicarea unor operatori asupra unor termeni care pot fi: constante, variabile, funcţii.

Expresiile se evaluează pe baza unui set de reguli care precizează prioritatea şi modul de asociere a operatorilor precum şi conversiile aplicate operanzilor:

• prioritatea determină ordinea de efectuare a operaţiilor într-o expresie cu diverşi operatori.

• modul de asociere indică ordinea de efectuare a operaţiilor într-o secvenţă de operaţii care au aceeaşi prioritate.

În tabela de mai jos se indică operatorii C++ în ordinea descrescătoare a priorităţii lor. Operatorii din aceeaşi categorie au aceeaşi prioritate. Operatorii de aceeaşi prioritate sunt prelucrati în ordinea de la stânga la dreapta sau la dreapta la stânga în direcţia indicată de sageată.

Categoria de operatori	Operatori	Prio-ritate	Mod de asociere
Primari: Apel de funcţie Indice de tablou Operator rezoluţie Referinţă la membru de structură Referinţă indirectă la membru structură	 ( ) [ ] :: . ->	15	→
Unari: Stabilirea tipului	 (tip)	14	

Categoria de operatori	Operatori	Prio-ritate	Mod de asociere
Dimensiune în octeţi Alocare memorie Dezalocare memorie Adresă Conţinut adresă Semn Negaţie Incrementare, decrementare	sizeof new delete & * + - ! ~ ++ --		←
Dereferenţierea pointerilor spre membrii claselor	.* ->*	13	→
Multiplicativi	* / %	12	→
Aditivi	+ -	11	→
Deplasare	<< >>	10	→
Relaţionali	< <= > >=	9	→
Egalitate	= = !=	8	→
ŞI la nivel de bit	&	7	→
SAU EXCLUSIV la nivel de bit	^	6	→
SAU la nivel de bit	|	5	→
ŞI logic	&&	4	→
SAU logic	||	3	→
Condiţional	?:	2	←
Atribuire	= += -= *= /= %= &= |= ^= <<= >>=	1	←
Operatorul virgulă	,	0	→

Operatori aritmetici

Operatorii + şi – unari se aplică unui singur operand şi se folosesc la stabilrea semnului operandului: pozitiv sau negativ.

Operatorul * reprezintă operatorul de înmulţire al operanzilor la care se aplică.

Operatorul / reprezintă operatorul de împărţire. Dacă ambii operanzi sunt întregi (char, int, unsigned, long), se realizează o împărţire întreagă, adică ne furnizează câtul împărţirii.

Operatorul % are ca rezultat restul împărţirii dintre doi operanzi întregi.

Operatorii binari + şi – reprezintă operaţiile obişnuite de adunare şi scădere.

Exemple:

int a, b;

float x, y;

Dacă a=3 şi b=7 atunci b/a are valoarea 2 iar b%a are valoarea 1.

Dacă x=9 şi y=2 atunci x/y are valoarea 4.5.

Expresia x*–y are sens, aici – este operatorul unar.

Operatori de incrementare / decrementare

Sunt operatori unari. Operandul asupra căruia se aplică trebuie să fie o variabilă întreagă sau flotantă. Operatorul de incrementare se notează prin ++, şi măreşte valoarea operandului cu 1, iar cel de decrementare se notează cu --, şi micşorează valoarea operandului cu 1.

Operatorii pot fi folosiţi prefixaţi:

```
++operand
--operand
```

sau postfixaţi:

```
operand++
operand--
```

În cazul în care sunt folosiţi postfixaţi, ei produc ca rezultat valoarea operandului şi apoi incrementează/decrementează operandul. Când se folosesc prefixaţi se incrementează/decrementează operandul după care produc ca rezultat valoarea incrementată/decrementată.

Exemple:

Expresie	Efect
j=i++	j=i; i=i+1;
y=--x	x=x-1; y=x;
x=v[3]--	x=v[3]; v[3]=v[3]-1;
x=++v[++j]	j=j+1; v[j]=v[j]+1; x=v[j];
y=++i-j	i=i+1; y=i-j;
y=i++-j	y=i-j; i=i+1;
y=(i-j)++	construcţie eronată

Operatori relaţionali

Operatorii relaţionali sunt `<, <=, >, >=, ==, !=`.

Rezultatul aplicării unui operator relaţional este 1 sau 0 după cum operanzii se află în relaţia definită de operatorul respectiv sau nu.

De exemplu, dacă a=5 şi b=6 atunci expresia a<=b are valoarea 1, iar expresia a+1>b are valoarea 0.

Operatorul == (egal) furnizează 1 dacă operanzii sunt egali şi zero în caz contrar. Operatorul != (*diferit*) furnizează 1 dacă operanzii nu sunt egali şi zero în caz contrar.

Exemple:

Dacă x=2 şi y =-1 atunci expresia x==y are valoarea 0, expresia x!=y are valoarea 1, expresia x+y==1 are valoarea 1.

Operatori logici

! – *negaţie logică*,operator unar.
&& – *ŞI logic*.
|| – *SAU logic*.

În limbajul C nu există valori logice speciale. Valoarea fals este reprezentată prin 0. Orice valoare diferită de 0 reprezintă valoarea adevărat. Operatorii logici admit operanzi de orice tip scalar. Rezultatul evaluării unei expresii logice este de tip întreg: zero pentu fals şi 1 pentru adevărat.

Dacă la evaluarea unei expresii logice se ajunge într-un punct în care se cunoaşte valoarea întregii expresii, atunci restul expresiei nu se mai evaluează.

Exemple:

Dacă a şi b sunt ambii diferiţi de zero expresia a&&b are valoarea 1, altfel 0.

Dacă a este negativ, expresia !(a<0) are valoarea 0, altfel 1.

Expresia !a&&b||a&&!b realizează SAU EXCLUSIV. Dacă a=0 şi b=1, deoarece !a&&b are valoarea 1 şi deci rezultatul întregii expresii este 1, subexpresia a&&!b nu se mai evaluează.

Operatori logici pe biţi

Constituie una din extensiile limbajului C spre limbajele de asamblare. Se aplică operanzilor de tip întreg, execuţia făcîndu-se bit cu bit, cu ajutorul celor patru operaţii logice şi două operaţii de deplasare la stînga şi la dreapta.

Se utilizează următoarele simboluri:

& pentru ŞI

| pentru SAU

^ pentru SAU EXCLUSIV

~ pentru NEGARE(complement faţă de unu, schimbă fiecare bit 1 al operandului în 0 şi fiecare bit 0 în 1)

<< pentru deplasare stânga

>> pentru deplasare dreapta

Cu excepţia operatorului negare care este unar, ceilalţi sunt operatori binari.

Operaţiile logice pentru o pereche oarecare de biţi x şi y se prezintă astfel:

x	y	x&y	x | y	x^ y	~ x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Exemple:Numerele x=5, y=36 se reprezintă în binar astfel:

x=00000101, y=00100100

```
        00000101                  00000101
        00100100                  00100100
x&y= ----------     ,    x | y = ----------
        00000100                  00100101

        00000101
        00100100                   00000101
x^y= ----------     ,      ~ x= ----------
        00100001                   11111010
```

Operatorul & se poate utiliza la anulări de biţi, de exemplu:

`a & 0x00ff` are ca valoare, valoarea octetului mai puţin semnificativ al valorii variabilei a (primii 8 biţi sunt înlocuiţi cu 0, iar următorii 8 biţi coincid cu cei ai lui a).

Operatorul | se poate utiliza la la setări de biţi, de exemplu:

`a | 0x00ff` primii 8 biţi ai rezultatului coincid cu cei ai lui a, iar următorii 8 sunt 1.

Operatorul <<, operator binar, realizează *deplasarea la stânga* a valorii primului operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său, biţii liberi din dreapta se completează cu zero. Această operaţie este echivalentă cu o înmulţire cu puteri ale lui 2.

Exemplu.

x=7 în baza 2 se scrie 0000 0000 0000 0111

x <<2 va produce 0000 0000 0001 1100 deplasarea cu o poziţie spre stânga şi adăugarea unui zero. Valoarea sa este 28 adica $7*2^2$

Operatorul >>, operator binar, realizează *deplasarea la dreapta* a valorii primului operand cu un număr de poziţii binare egal cu valoarea celui de-al doilea operand al său. Această operaţie este echivalentă cu o împărţire cu puteri ale lui 2.

In cazul deplasarii spre dreapta, biţii liberi din stânga se completează automat cu zero numai dacă numărul este nenegativ. Dacă numărul este negativ, din necesitatea de a conserva semnul (reprezentat în bitul cel mai semnificativ cu 1), biţii liberi din stînga se completează cu 1.

Exemple:

19>>3 are ca rezultat numărul binar 0000 0000 0000 0010 egal cu 2 adică $19/2^3$.

x=-9 se reprezintă în binar în cod complementar faţă de 2. Acesta se obţine adunând la complementul faţă de unu al numărului, valoarea 1.

Deci – 9 se obţine ca ~ 9+1.

9 se reprezintă prin:	0000 0000 0000 1001
~ 9 se repretintă prin:	1111 1111 1111 0110
~ 9 +1 se reprezintă prin:	1111 1111 1111 0111
atunci – 9 se reprezintă prin:	1111 1111 1111 0111
x>>2 va produce:	1111 1111 1111 1101

Operatori de atribuire

Operatorul = se utilizează în construcţii de forma `v = expresie`.

Această construcţie se numeşte expresie de atribuire şi este un caz particular de expresie. Tipul ei coincide cu tipul lui v, iar valoarea ei este chiar valoarea atribuită lui v. Rezultă că o expresie de forma:

```
v1 = (v = expresie)
```

este legală.

Deoarece operatorii de atribuire se evaluează de la dreapta la stânga expresia de mai sus se poate scrie fără paranteze.

În general, putem realiza atribuiri multiple de forma

$v_n=v_{n-1}=...=v_1$=expresie.

În cazul unei expresii de atribuire, dacă expresia din dreapta semnului egal are un tip diferit de cel al variabilei v, atunci întâi se converteşte valoarea ei spre tipul variabilei v şi pe urmă se realizează atribuirea.

Pentru operaţia de atribuire putem folosi operatorii de atribuire combinată:

```
op=
```

Unde prin op se înţelege unul din operatorii binari aritmetici sau logici pe biţi, adică: *, /, %, +, -, &, |, ^, <<, >>.

Expresia:

```
v op= expresie
```

este echivalentă cu

```
v = v op (expresie)
```

Se pot realiza maximum zece combinaţii:

```
+= ,-=, *=, /=, %=, & =, |=, ^=, <<=, >>=
```

Exemple:

Expresia: `x=x+5` este echivalentă `cu x+=5`.

Expresia `x=x^y` este echivalentă cu `x^=y`.

Expresia `x=x<<y` este echivalentă cu `x<<=y`.

Expresia: `x/=y+3` este echivalentă cu `x=x/(y+3)`.

Expresia: `tab[i*j+1]=tab[i*j+1]*z` este echivalentă cu `tab[i*j+1]*=z`.

Expresia `C=C*n/k`, cu `C`, `n` şi `k` de tip întreg, nu produce acelaşi rezultat cu `C*=n/k`, de exemplu pentru C=20, n=5, k=2 prima expresie furnizează rezultatul 50, iar a doua 40 (`C*=n/k` este echivalentă cu `C=C*(n/k)`).

Operatorul de conversie explicită

Dacă dorim să forţăm tipul unui operand sau al unei expresii putem folosi o construcţie de forma `(tip)operand`. Prin aceasta, valoarea operandului se converteşte spre tipul indicat în paranteze.

În C++, dacă tip este format dintr-un singur cuvânt, se poate folosi şi construcţia: `tip(operand)`

Exemple:

```
int x,y;
double z;
x=10; y=4;
 z=x/y;   /* z primeşte valoarea 2.0 deoarece împărţirea, făcându-se
între operanzi de tip întreg,
        este o împărţire întreagă*/
z=(double)x/(double)y; // rezultatul va fi z=2.5.
```

Construcţia: `(float)(x+y)` este echivalentă în C++ cu `float(x+y)`.

Construcţia: `unsigned char(x)` este eronată deoarece tipul conversiei este format din două cuvinte.

Operatori condiţionali

Operatorii condiţionali se utilizează în evaluări de expresii care prezintă alternative. Ei sunt „`?:`”.

O astfel de expresie are formatul:

```
exp1 ? exp2 : exp3
```

şi are următoarea interpretare: dacă `exp1` este diferită de zero, atunci valoarea şi tipul expresiei condiţionale sunt date de valoarea şi tipul expresiei `exp2` altfel de valoarea şi tipul lui `exp3`.

Exemplu:

```
y?x/y:x*x
```

Maximul dintre două numere se poate determina astfel:

```
max=a>b?a:b;
```

Operatorul virgulă

Există cazuri în care este util să grupăm mai multe expresii într-una singură, expresii care să se evalueze succesiv. În acest scop se foloseşte operatorul virgulă care separă secvenţa de expresii, acestea grupându-se într-o singură expresie.

Expresia:

exp_1, exp_2, ... , exp_n

va avea ca valoare şi tip valoarea şi tipul lui exp_n, deci cu a ultimei expresii.

Exemple:

```
k=(x=10, y=2*i-5, z=3*j, x+y+z);
```

Se execută pe rând cele trei atribuiri, apoi se efectuează suma x+y+z care se atribuie lui k.

```
++i, --j
```

i se măreşte cu o unitate, j se micşorează cu o unitate, valoarea şi tipul întregii expresii coincid cu valoarea şi tipul lui j.

Operatorul dimensiune (sizeof)

Lungimea în octeţi a unei date se poate afla cu o construcţie de forma

```
sizeof(data)
```

unde data este numele unei variabile, al unui tablou, al unui tip etc.

Exemple:

```
int i;
float x;
char c;
double d;
int tab[10];

sizeof(i)              -furnizează valoarea 2
sizeof(x)              -furnizează valoarea 4
sizeof(float)          -furnizează valoarea 4
sizeof(c)              -furnizează valoarea 1
sizeof(tab)            -furnizează valoarea 20
sizeof(tab[i])         -furnizează valoarea 2
```

Operatori paranteză

Operatorul paranteză rotundă „()” se utilizează pentru a impune o altă ordine în efectuarea operaţiilor. O expresie inclusă între paranteze rotunde formează un operand.

Parantezele rotunde se utilizează şi la apelul funcţiilor.

Parantezele pătrate „[]” includ expresii care reprezintă indici. Ele formează operatorul de indexare.

Operatori de adresă

Operatorul & returnează adresa unei variabile. Astfel, dacă x este o variabilă, &x va fi adresa variabilei x.

Operatorul * returnează valoarea de la o anumită adresă. Astfel, dacă p este pointer la tipul char, *p va fi caracterul referit de p.

Regula conversiilor implicite

Dacă operanzii unui operator binar nu sunt de acelaşi tip, sunt necesare conversii care se execută automat astfel:

1. Fiecare operand de tip char, unsigned char sau short se converteşte spre tipul int şi orice operand de tip float se converteşte spre tipul double.
2. Dacă unul dintre operanzi este de tip long double, atunci şi celălalt se converteşte la tipul long double şi rezultatul va fi de tip long double.
3. Dacă unul dintre operanzi este de tip double, atunci şi celălalt se converteşte la tipul double şi rezultatul va fi de tip double.
4. Dacă unul dintre operanzi este de tip unsigned long, atunci şi celălalt se converteşte la tipul unsigned long şi rezultatul va fi de tip unsigned long.
5. Dacă unul dintre operanzi este de tip long, atunci şi celălalt se converteşte la tipul long şi rezultatul va fi de tip long.
6. Dacă unul dintre operanzi este de tip unsigned, atunci şi celălalt se converteşte la tipul unsigned şi rezultatul va fi de tip unsigned.

I.2. Structura programelor C

Prin *program* înţelegem un text ce specifică acţiuni ce vor fi executate de un procesor. Limajul C este un limbaj procedural ceea ce înseamnă că structura programelor scrise în C se bazează pe subprograme.

Un *subprogram* este o secvenţă de declaraţii şi instrucţiuni care formează o structură unitară, ce rezolvă o problemă de complexitate redusă, putând fi innclus într-un program sau stocat în biblioteci, compilat separat şi utilizat ori de câte ori este nevoie.

Acţiunile sunt descrise cu ajutorul instrucţiunilor.

In limbajul C subprogramele sunt realizate sub formă de funcţii.

Un program C se compune din una sau mai multe funcţii. Fiecare funcţie are un nume. Orice program scris în limbajul C, indiferent de complexitate, trebuie să conţină o funcţie, numită funcţie principală, al cărui nume este *main*.

Această funcţie preia controlul de la sistemul de operare în momentul în care programul este lansat în execuţie şi îl redă la terminarea execuţiei.

Execuţia unui program C înseamnă execuţia instrucţiunilor din funcţia `main`.

Structura unei funcţii este următoarea:

```
<tip> <nume>( <lista parametrilor formali>)    //antet
  {
   <declaraţii şi instrucţiuni>      //corpul funcţiei
  }
```

Structura generală a unui program C este următoarea:

```
<directive preprocesor>

<declaratii globale>

<tip> <functie1> (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }
  ...

<tip> <functien> (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }

<tip> main (<listă parametri>)
  {
    <declaraţii locale si instrucţiuni>
  }
```

Un program C parcurge următoarele etape:

- *editarea fişierului sursă,* care constă în scrierea programului folosind regulile de sintaxă ale editorului de texte corespunzător;
- *compilarea fişierului sursă,* un program specializat numit compilator, transformă instrucţiunile programului sursă în instrucţiuni maşină. Dacă nu detectează erori sintactice, el va genera un fişier numit fişier obiect, care are numele fişierului sursă şi extensia .obj.
- *editarea legăturilor,* un program specializat numit linkeditor, asamblează mai multe module obiect şi generează programul executabil sub forma unui fişier cu extensia .exe. Pot să apară erori generate de incompatibilitatea modulelor obiect asamblate.
- *lansarea în execuţie* – programul se află sub formă executabilă şi poate fi lansat în execuţie. În această etapă pot să apară erori fie datorită datelor eronate introduse în calculator, fie conceperii greşite a programului.

În limbajul C există două categorii de funcţii:

• funcţii care produc (returnează) un rezultat direct ce poate fi utilizat în diverse expresii. Tipul acestui rezultat se defineşte prin <tip> din antetul funcţiei. Dacă <tip> este absent, se presupune că funcţia returnează o valoare de tip int.

• funcţii care nu produc un rezultat direct. Pentru aceste funcţii se va folosi cuvântul cheie `void` în calitate de tip. El semnifică lipsa valorii returnate la revenirea din funcţie.

O funcţie poate avea zero sau mai mulţi parametri separaţi prin virgule.

Dacă o funcţie are lista parametrilor formali vidă, antetul său se reduce la:

```
<tip> <nume>()
```

Absenţa parametrilor formali poate fi indicată explicit folosind cuvântul cheie *void*. Astfel, antetul de mai sus poate fi scris şi sub forma:

```
<tip> <nume>(void)
```

Exemple:

```
int g(float x, long n)
  {
    ...
  }

void f(void)
  {
   ...
  }

void main()
  {...}
```

I.2.1. Preprocesare

Un program C poate suporta anumite prelucrări înainte de compilare. O astfel de prelucrare se numeşte preprocesare. Ea se realizează printr-un program special numit preprocesor. Preprocesorul este apelat automat înainte de a începe compilarea.

Prin intermediul preprocesorului se pot realiza:

• includeri de fişiere standard şi utilizator;

• definirea de macrodefiniţii;

• compilare condiţionată.

I.2.2. Includeri de fişiere

Fişierele se includ cu ajutorul construcţiei #include folosindu-se formatele:

```
#include<specificator_de_fişier>
#include "specificator_de_fişier"
```

Preprocesorul localizează fişierul şi înlocuie construcţia include cu textul fişierului localizat. În felul acesta compilatorul C nu va mai întâlni linia `#include`, ci textul fişierului inclus de preprocesor.

Prima variantă se foloseşte pentru încorporarea fişierelor standard ce se găsesc în bibliotecile ataşate mediului de programare. A doua variantă se foloseşte uzual pentru încorporarea fişierelor create de utilizator; dacă nu este specificată calea atunci fişierul este căutat în directorul curent şi în bibliotecile ataşate mediului de programare.

Includerile de fişiere se fac, de obicei, la începutul fişierului sursă. Textul unui fişier inclus poate să conţină construcţia `#include` în vederea includerii altor fişiere.

Exemple:

```
#include<iostream.h>
#include"geo.cpp"
```

I.2.3. Macrodefiniţii

O altă construcţie tratată de preprocesor este construcţia `define` cu formatul:

```
#define <nume> <succesiune de caractere>
```

Folosind această construcţie, preprocesorul substituie `<nume>` cu `<succesiune de caractere>` peste tot în textul sursă care urmează, exceptând cazul în care `<nume>` apare într-un şir de caractere sau într-un comentariu. Dacă succesiunea de caractere nu încape pe un rând ea poate fi continuată terminând rândul cu "\".

Se recomandă ca `<nume>` să se scrie cu litere mari; <succesiune de caractere> poate conţine alte macrodefiniţii care trebuie să fie în prealabil definite.

O macrodefiniţie este definită din punctul construcţiei `#define` şi până la sfârşitul fişierului sursă respectiv sau până la redefinirea ei sau până la anihilarea ei prin intermediul construcţiei: `#undef <nume>`

Exemple:

```
#define PI 3.14159
#define DIM 100
#define A 123
#define B A+120
...
x=3*B      // se substituie prin x=3*123+120
#define A 123
#define B (A+120)
...
x=3*B      // se substituie prin x=3*(123+120)
```

Macrodefiniţii cu argumente

Directiva `#define` poate fi folosită şi în sintaxa:

```
#define <nume>(<listă de parametri>) <corp macrodefinţie>
```

între <nume> şi "(" nu există spaţii.

Exemplu:

```
#define MAX(a,b) ((a)>(b)?(a):(b))
...
x=max(k+5,m)
```

I.2.4. Compilare condiţionată

Compilarea condiţionată se realizează folosind construcţiile:

1.
```
#if  expresieConstantă  //dacă expresie este diferită de zero
  text
#endif
```
2.
```
#if  expresieConstantă
   text1
#else
   text2
#endif
```
3.
```
#ifdef  identificator //dacă identificator a apărut într-o directivă #define
   text
#endif
```
4.
```
#ifdef  identificator // dacă identificator a apărut într-o directivă #define
   text1
#else
   text2
#endif
```
5.
```
#ifndef  identificator //dacă identificator nu a apărut într-o directivă #define
   text
#endif
```
6.
```
#ifndef  identificator //dacă identificator nu a apărut într-o directivă  #define
   text1
#else
   text2
#endif
```

Pentru toate directivele `if` liniile care urmează până la o directivă `#endif` sau `#else` sunt supuse preprocesării dacă condiţia testată este satisfăcută şi sunt ignorate dacă condiţia nu este satisfăcută. Liniile dintre `#else` şi `#endif` sunt supuse preprocesării dacă condiţia testată de directiva #if nu este satisfăcută.

Exemplu:

```
#ifndef tipData
```

```
   #define tipData long
#endif
tipData x;
```

I.3. Operaţii de intrare-ieşire

Operaţiile de intrare/ieşire asigură instrumentul necesar pentru comunicarea dintre utilizator şi calculator. LIMBAJUL C/C++ asigură posibilitatea de a folosi orice echipamente periferice: consola (tastatura şi ecranul), imprimanta, unităţi de magnetice diverse etc.

Pentru uniformizarea modului de lucru cu dispozitivele de intrare/ieşire, se introduce un nivel intermediar între program şi echipamentul periferic folosit. Forma intermediară a informaţiilor corespunde unui echipament "logic" şi se numeşte "flux" sau "şir" de informaţii (stream, în limba engleză) şi nu depinde de echipamentul periferic folosit. Un flux de informaţii constă dintr-o succesiune ordonată de octeţi şi poate să fie privit ca un tablou unidimensional de caractere de lungime neprecizată. Citirea sau scrierea la un echipament periferic constă în citirea datelor dintr-un flux, sau scrierea datelor într-un flux.

I.3.1. Operaţii de intrare / ieşire, utilizând consola, în C++

Limbajul C++ nu dispune de instrucţiuni specifice pentru operaţiile de intrare/ieşire. Acestea se efectuează cu ajutorul unor funcţii de bibliotecă ce folosesc conceptele generale ale limbajului C++: programare orientată pe obiecte, cu ierarhii de clase, moşteniri multiple, supraîncărcarea operatorilor.

Fişierul `iostream.h`, care trebuie inclus în orice program pentru a apela operaţiile de I/E specifice C++, conţine cele mai importante funcţii de lucru cu tastatura şi ecranul.

De asemenea în acest fişier sunt definite fluxurile:

`cin` folosit pentru intrare, dispoziţiv implicit tastatura (*console input*);

`cout` folosit pentru ieşire, dispoziţiv implicit ecranul (*console output*);

`cerr` folosit pentru afişarea erorilor, dispoziţiv implicit ecranul;

`clog` folosit pentru afişarea erorilor, dispozitiv implicit ecranul; `clog` reprezintă versiunea cu tampon a lui `cerr`.

Deoarece conceptele programării orientate pe obiecte nu au fost prezentate, vom explica numai modul de utilizare a funcţiilor de I/E.

Vom accepta că în fişierul `iostream.h` sunt definite noi tipuri de date printre care `istream` şi `ostream`. Obiectele de tipul `istream` sunt dispozitive logice de intrare (`cin` este un astfel de obiect).

Obiectele de tipul `ostream` reprezintă dispozitive logice de ieşire (`cout, cerr, clog` sunt astfel de obiecte).

Se prevăd două nivele de interfaţă între programator şi dipozitivele logice de intrare/ieşire, prin două seturi de funcţii:

a) funcţii pentru operaţii de I/E la nivel înalt

b) funcţii pentru operaţii la nivel de caracter.

I.3.2. Funcţii pentru operaţii de I/E la nivel înalt

Pentru operaţiile de I/E la nivel înalt, au fost supraîncărcaţi operatorii ">>" pentru fluxul *cin* şi "<<" pentru fluxul *cout*, *cerr*, *clog*.

Operatorul ">>", redefinit, se numeşte *operator de extracţie* sau *extractor*. Această denumire provine de la faptul că la citire se extrag date dintr-un stream.

Operatorul "<<", redefinit, se numeşte *operator de inserţie* sau *insertor*. Această denumire provine de la faptul că un operator de ieşire inserează informaţia în stream.

Ambii operatori posedă proprietatea de asociativitate la stânga şi returnează o referinţă la streamul asociat ca prim operand (cin pentru ">>", respectiv cout, cerr sau clog pentru "<<"), deci ei pot fi înlănţuiţi.

Exemplu:

```
#include<iostream.h>
void main()
  { float v,t,d;
    cout<<”Distanta parcursa(Km)=”;  cin>>d;
    cout<<”Numar de ore =”;          cin>>t;
    v=d/t;
    cout<<”Viteza medie este de ”<<v<<” km/h”;
         // înlănţuirea operatorului <<
   }
```

Nu se acceptă înlănţuiri mixte ("<<" cu ">>").

Faptul ca operatorii returnează o referinţă către stream, permite să se poată testa starea streamului ca în exemplul următor:

```
#include <iostream.h>
void main()
  { int n;
    if(cin>>n) cout<<"citire cu succes";
    else  cout<<"esec la citire";  // s-a tastat un şir nenumeric
  }
```

Tipurile de date predefinite acceptate implicit de un stream de intrare sunt: char, short, int, long toate cu sau fără unsigned, float double şi şir de caractere.

A extrage o valoare numerică dintr-un şir înseamnă:

• ignorarea tuturor caracterelor de tip spaţiu;

• un test dacă primul caracter nespaţiu este cifră, semn sau punct zecimal (pentru tipurile flotante). În caz că această condiţie este îndeplinită se va continua cu următorul pas, altfel se va semnaliza eroare şi nu se vor mai putea efectua operaţii de intrare până ce nu se tratează respectiva eroare;

• se extrag toate caracterele consecutive, până la întâlnirea unuia invalid pentru respectivul tip de dată;

• în cele din urmă, se va efectua conversia şirului extras la tipul de dată respectiv.

În cazul tipului de dată char, o secvenţă de genul:

```
char c;
cin>>c;
```

atribuie variabilei c, primul caracter nespaţiu aflat în streamul de intrare.

Dacă se doreşte extragerea unui şir de caractere, se vor citi toate caracterele întâlnite consecutiv în streamul de intrare, începând cu primul caracter nespaţiu din stream, până se va ajunge la un caracter spaţiu. Stringului astfel obţinut i se va adăuga la sfârşit caracterul NULL (\0).

Să presupunem că, la o operaţie de citire, primul caracter nespaţiu din streamul de intrare va fi invalid pentru tipul de dată citit. În acest caz valoarea variabilei destinaţie va rămâne neschimbată. În plus se va seta un indicator, indicatorul "fail" (eşec) al streamului de intrare, fapt ce va duce la imposibilitatea de a mai citi din acest stream până la resetarea indicatorului.

Indicatorul de eroare se anulează cu ajutorul metodei (funcţiei) `clear()`.

Sintaxa de apel pentru streamul `cin` este:

```
cin.clear();
```

Metoda fail(), returnează o valoare non-zero (true) dacă streamul de citire se află în "stare de eroare" şi 0 dacă citirea s-a efectuat cu succes.

Sintaxa de apel pentru streamul `cin` este:

```
cin.fail()
```

Metoda `eof()`, returnează o valoare 1 (true) dacă s-a atins sfârşitul streamului din care citim şi 0 în caz contrar.

Sintaxa de apel pentru streamul `cin` este:

```
cin.eof()
```

Tipurile de date predefinite pe care le acceptă implicit un stream de ieşire sunt: `char, short, int, long` cu sau fără `unsigned, float, double, long double`, pointeri la aceste tipuri şi tipul `void*`.

Pointerul la caracter `(char*)` este utilizat la tipărirea şirurilor de caractere iar `void*` la cea a variabilelor de tip pointer (afişarea se face în hexa).

Exemple:

1.

```
#include <iostream.h>
void main()
  { char c;
    long n;
    float x;
    double t;
    cin>>c; cout<<"c="<<c<<'\n'; // cout<<'\n' are ca efect saltul la linie nouă.
    cin>>n; cout<<"n="<<n<<'\n';
    cin>>x; cout<<"x="<<x<<'\n';
    cin>>t; cout<<"t="<<t<<'\n';
  }
```

2.

```
#include <iostream.h>
void main()
  { char s[20];
    cin>>s; cout<<s;
  }
```

Deoarece numele unui tablou are ca valoare adresa primului său element, înseamnă că, în exemplul de mai sus, s este de fapt un pointer (constant) către caractere.

3.

```
#include <iostream.h>
void main()
  { char* a="xyzw";
    cout<<"sirul a="<<a<<" adresa sirului a="<<(void*)a;
  }
```

I.3.3. Manipulatori de intrare / ieşire

Pentru situaţia în care nu dorim ca citirea dintr-un stream sau scrierea într-un stream să se facă în formatele implicite din C++, sunt prevăzute şi posibilităţi de a controla formatele de intrare şi de ieşire prin comenzi incluse în construcţiile de intrare/ieşire.

Aceste comenzi se numesc manipulatori de I/E.

Pentru a putea folosi manipulatorii care au parametri (de exemplu setw()) trebuie să includem fişierul iomanip.h. Acest lucru nu este necesar dacă utilizăm manipulatori fără parametri. Manipulatorii pot apare în lanţul de operaţii de I/E.

Lista manipulatorilor de I/E este următoarea:

Manipulator	Scop	Intrare/ ieşire
dec	formatează datele numerice în zecimal valabil până la resetare	E
oct	formatează datele numerice în octal, valabil până la resetare	E
hex	formatează datele numerice în hexazecimal, valabil până la resetare	E
setbase(int baza)	stabileşte baza de numeraţie la *baza* (8,10,16), valabil până la resetare	E
setw(int w)	stabileşte la w numărul de poziţii pe care se va afişa următoarea dată de scris.	E
setprecision(int p)	stabileşte numărul de cifre aflat după punctul zecimal. comanda este valabilă până la reapelare.	E
setfill(int ch)	stabileşte caracterul de umplere, implicit spaţiu, valabil până la resetare	E
setiosflags(long f)	activează indicatorii de format specificaţi în variabila f.	I/E
resetiosflags(long f)	dezactivează indicatorii de format specificaţi în variabila f.	I/E
flush	eliberează un stream	E
endl	scrie un caracter "newline" şi eliberează streamul	E

Manipulator	Scop	Intrare/ ieşire
ends	Scrie un caracter null	E
ws	ignoră caracterele de tip spaţiu	I

Exemplu:

```
#include<iostream.h>
#include<iomanip.h>
void main()
  { cout<<hex<<100<<endl;
    cout<<dec <<setfill('*')<<setw(5)<<100<<endl;
  }
```

I.3.4. Indicatori de format

Fiecare stream din C++ are asociat un număr de indicatori de format flags. Ei sunt codificaţi într-o variabilă de tip *long*.

Următoarele constante definesc aceşti indicatori de format:

Constanta	Valoare	Rol
ios::skipws	0x0001	Ignoră caracterele de tip spaţiu de la intrare
ios::left	0x0002	aliniere la stânga în ieşire
ios::right	0x0004	aliniere la dreapta în ieşire
ios::internal	0x0008	semn aliniat ex: –55 —> – 55
ios::dec	0x0010	conversie în baza 10
ios::oct	0x0020	conversie în baza 8
ios::hex	0x0040	conversie în hexazecimal
ios::showbase	0x0080	se va tipări şi baza (0 - octal, 0x -hexa)
ios::showpoint	0x0100	afişează şi zerourile nesemnificative de după punctul zecimal
ios::uppercase	0x0200	Pentru "literele" din baza 16 se vor utiliza majuscule
ios::showpos	0x0400	întregii pozitivi sunt prefixati de "+"
ios::scientific	0x0800	Pentru date flotante se foloseşte notaţia stiinţifică (1.234e2)
ios::fixed	0x1000	utilizează notaţia normală (123.45)
ios::unitbuf	0x2000	streamul se goleşte (afişează) după fiecare inserare
ios::stdio	0x4000	Stream-urile predefinite de ieşire se vor goli după fiecare inserare

Prin aplicarea operatorului de tip or "|" putem poziţiona mai mulţi indicatori.

De exemplu:

```
cout<<setiosflags(ios::left|ios::showpoint);
```

realizează alinierea la stânga şi afişarea zerourilor nesemnificative de după punctul zecimal.

Pentru stream-urile standard, avem următoarele valori implicite:
pentru `cin: 0x0001` pentru `cout: 0x2001`
pentru `cerr: 0x2001` pentru `clog: 0x0001`
Pentru a afla stările curente ale indicatorilor, se foloseşte funcţia `flags()`.
Funcţia se apelează utilizând formatul: `<stream>.flags()` şi returnează o valoare de tip `long` ce conţine, codificat, indicatorii de stare ai streamului asociat.

Exemplu: Indicatorii streamului cin se pot afişa astfel:

```
cout<<hex<<cin.flags();
```

I.3.5. Funcţii pentru operaţii la nivel de caracter

Funcţiile uzuale la nivel de caracter, asociate unui stream sunt:

1) `stream& put(char c)`

– inserează un caracter în streamul de ieşire şi returnează streamul de ieşire;
Exemplu de apel:

```
char x;
...
cout.put(x);
```

2) `int get(void)`

– extrage următorul caracter din streamul de intrare (chiar spaţiu) şi îl returnează (se returnează EOF(-1) când se ajunge la sfârşitul intrării);
Exemplu de apel:

```
char x;
x=cin.get();
```

Exemplu de utilizare:
Următorul program copiază caracterele din streamul de intrare cin (până la întâlnirea caracterului CTRL Z) în streamul de ieşire cout.

```
#include <iostream.h>
void main()
  { char c;
    while((c=cin.get())!=EOF) cout.put(c);
  }
```

3) `istream& get(unsigned char&)`
4) `istream& get( signed char& )`

Ambele funcţii extrag următorul caracter din streamul de intrare în variabila dată ca argument şi returnează streamul de intrare.
Exemplu de apel:

```
char x;
cin.get(x);
```

5) `istream& get (unsigned char *c, int n, char='\n');`
6) `istream& get ( signed char *c, int n, char='\n');`

Ambele funcţii extrag un şir de caractere din streamul de intrare în variabila tablou "c". Numărul maxim de caractere al şirului este dat de al doilea argument (se extrag maxim n–1 caractere).

Al treilea argument indică aşa-zisul caracter de sfârşit. Caracterul de sfârşit nu va fi citit şi nici nu va fi eliminat din stream.

Exemple de apel:

```
char c[10];
cin.get(c,10);
     //se extrag 9 caractere sau până la întâlnirea caracterului de sfârşit implicit '\n'
cin.get(c,10,',');
     //se extrag 9 caractere sau până la întâlnirea caracterului "," care nu va fi extras din şir
```

7) `istream& getline(unsigned char *c, int n,  char='\n');`
8) `istream& getline(signed char *c, int n, char='\n');`

Sunt echivalente cu `get` dar extrag (fără a fi depus în `c`), eventual, şi delimitatorul dacă este întâlnit înainte de a citi toate cele n–1 caractere propriu-zise ale şirului.

Exemple de apel:

```
char c[10];
cin.getline(c,5);
```

- se extrag şi se depun în c, 4 caractere sau până la întâlnirea caracterului '\ n' (<enter>), care va fi extras din stream.

```
cin.getline(c,5,'.');
```

- se extrag 4 caractere sau până la întâlnirea caracterului '.' care va fi extras din stream.

9) `istream& ignore(int n=1, int =EOF);`

Neglijează un număr maxim de caractere, dat de primul argument (valoare implicită 1) sau până la întâlnirea delimitatorului dat de al doilea argument. Returnează o referinţă către streamul de intrare.

Exemple de apel:

```
cin.ignore(80,'#'); // ignoră următoarele 80 de caractere sau până întâlneşte "#"
cin.ignore();          //ignoră următorul caracter.
```

Exemplu de utilizare combinată a metodelor eof(), fail(), ignore():

```
#include <iostream.h>
void main()
 {int x,i;
  i = 0;
  while (1) // ciclu infinit; se iese din el la tastarea combinatiei CTRL+Z
```

```
   {i++;
    cout<<"x"<<i<<"="; cin >> x;
    if(cin.eof()) break; //s-a atins sfarsitul streamului (CTRL+Z)
    if (cin.fail()) //esec la citire
     { cout <<"eroare"<<endl; i--;
       cin.clear(); // resetam indicatorul de eroare
       cin.ignore(80,'\n'); /* eliminam din stream caracterele
                                                  care au generat eoarea */
     }
    }
  }
```

10) `int gcount();`

Returnează numărul de caractere citite din stream la ultima operaţie de citire.

Exemplu de apel:

```
void main()
{ int x; char y[25];
  cin>>x>>y;
  cout<<cin.gcount();
}
```

Dacă tastăm: `1234 abc<enter>` rezultatul va fi 3.

11) `int peek();`

- întoarce caracterul următor fără să-l extragă din stream

12) `istream& putback(char c);`

- inserează caracterul c în stream (ca prim caracter de citit)

```
#include <iostream.h>
void main()
   { char c;
      c=cin.get();       // citeşte primul caracter din stream
      cout<<c<<endl;     // afişează caracterul citit
      cin.putback('x'); // inserează 'x' în stream
      c=cin.get();       // citeşte caracterul inserat
      cout<<c<<endl;
   }
```

13) `istream& read(char * c, int n);`

- citeşte n caractere din stream în tabloul *c*. Niciun caracter de delimitare nu întrerupe citirea.

Exemplu:

```
#include <iostream.h>
void main()
 { char c[10];
   cin.read(c,5); //citeşte 5 caractere la care adaugă caracterul '\0'
   cout<<c<<endl;
 }
```

14) `ostream& write(char *c, int n);`

- inserează *n* caractere din şirul *c* în streamul de ieşire asociat

I.3.6. Gestiunea ecranului în mod text

Biblioteca standard a limbajului C /C++ conţine funcţii pentru gestiunea ecranului. Acesta poate fi gestionat în două moduri: mod text sau mod grafic.

Funcţiile standard de gestiune a ecranului în mod text au prototipurile în fişierul `conio.h`.

În mod curent ecranul este format din 25 linii şi 80 coloane. Colţul stânga sus al ecranului are coordonatele (1,1). Colţul dreapta jos al ecranului are în mod curent coordonatele (80,25).

Prezentăm câteva funcţii, uzuale, pentru gestiunea ecranului:

15) `void window(int x1,int y1,int x2,int y2);`

- Defineşte o fereastră pe ecran, unde (x1,y1) reprezintă colţul stânga sus al ferestrei, (x2,y2) reprezintă colţul dreapta jos al ferestrei.

Funcţiile de gestiune ale ecranului acţionează numai asupra ferestrei active. Dacă parametrii de apel sunt eronaţi, funcţia nu are efect.

Exemplu:

```
#include <iostream.h>
#include <conio.h>
void main()
  { clrscr();
    window(40,10,60,20);
    cout<<"xxx";
    cout<<wherey()<<wherex();
  }
```

16) `void clrscr(void);`

- şterge fereastra activă şi poziţionează cursorul în colţul stânga sus al ferestrei active, adică în poziţia de coordonate (1,1).

17) `int wherex(void);`

- returnează numărul coloanei pe care se află cursorul

18) `int wherey(void);`

- returnează numărul liniei pe care se află cursorul

19) `void gotoxy(int x,int y);`

- mută cursorul în punctul de coordonate (x,y) relative la fereastra activă.

În fişierul `conio.h` există şi funcţii de lucru cu tastatura, printre care:

20) `int getche();`

21) `int getch();`

- realizează citirea de la intrarea standard a caracterului curent şi returnează codul ASCII al caracterului citit. Aceste funcţii au acces direct la caracter de îndată ce acesta a fost tastat. getche() efectuează citirea cu ecou, pe când getch() o efectuează fără ecou.

```
22) int kbhit(void);
```

- funcţia întoarce o valoare diferită de 0 dacă a fost apăsată o tastă.

I.4. Instrucţiuni C/C++

Descrierea acţiunilor ce vor fi executate de calculator se face cu ajutorul instrucţiuilor.

I.4.1. Instrucţiunea de atribuire

Se obţine scriind punct şi virgulă după o expresie de atribuire sau după o expresie în care se aplică la o variabilă unul din operatorii de incrementare sau decrementare. Deci o instrucţiune de atribuire are una din următoarele formate:

```
<expresie de atribuire>;
<variabila>++;
++<variabila>;
<variabila>--;
--<variabila>;
```

I.4.2. Instrucţiunea compusă (blocul)

Instrucţiunea compusă este o succesiune de instrucţiuni incluse între acolade, succesiune care eventual poate conţine declaraţii. Sintaxa:

```
{
  <declaraţii şi instrucţiuni>
}
```

Dacă sunt prezente, declaraţiile definesc variabile care sunt valabile numai în instrucţiunea compusă respectivă. După paranteza închisă a unei instrucţiuni compuse nu se pune ";"

Structura unei funcţii poate fi considerată ca fiind:

```
<antetul funcţiei>
   <instrucţiune compusă>
```

I.4.3. Instrucţiunea if

Instrucţiunea if implementează structura alternativă.
Ea are unul din formatele:

Formatul 1:

```
if(<expresie>)
   <instrucţiune1>
else
   <instrucţiune2>
```

Dacă <*expresie*> este diferită de zero se execută <*instrucţiune1*>, altfel se execută <*instrucţiune2*>.

Formatul 2:

```
if(<expresie>) <instrucţiune>
```

Dacă <*expresie*> este diferită de zero se execută <*instrucţiune*>, altfel instrucţiunea if nu are niciun efect.

O selecţie multiplă se poate programa cu mai multe instrucţiuni if – else în cascadă.

Sintaxa:

```
if ( <expresie1> ) <instrucţiune1> ;
else if ( <expresie2> ) <instrucţiune2> ;
 . . .
     else if ( <expresien> ) <instrucţiunen> ;
          else <instrucţiunen+1> ;
```

Trebuie ţinut seama de faptul că *else* este asociat celui mai apropiat if.

Pentru a determina un alt mod de asociere se pot utiliza delimitatorii de bloc.

```
if ( <expresie1> )
  { if ( <expresie2 > ) <instrucţiune1 >  }
else <instrucţiune2 >
```

În acest mod *else* se asociază primului *if* şi nu celui de–al doilea care este mai apropiat.

Exemplu de utilizare a instrucţiunii if:

Programul următor calculează valoarea funcţiei

$$f(x)=\begin{cases}4x^2+2x-1, \text{ dacă } x<0\\ 50, \text{ dacă } x=0\\ 2x^2+8x+1, \text{ dacă } x>0\end{cases}$$

într-un punct x introdus de la tastatură.

```
#include<iostream.h>
#include<conio.h>

void main()
  { float x,f;
```

```
    cout<<"x="; cin>>x; // Citirea lui x

      // Evaluarea functiei:
    if(x<0) f=4*x*x+2*x-1;
    else if(x==0)  f=50;
         else      f=2*x*x+8*x+1;
    cout<<"f("<<x<<")="<<f<<endl;     // Afisarea rezultatului
}
```

I.4.4. Instrucţiunea while

Instrucţiunea `while` implementează structura repetitivă cu test iniţial şi are sintaxa:

```
while(<expresie>)
    <instrucţiune>
```

Se execută <instrucţiune> cât timp <*expresie*> este adevărată.

Exemple de utilizare a instrucţiunii while:

1) Fie x un vector cu n elemente flotante ce se introduc de la tastatură. Următorul program inversează ordinea elementelor în vector.

```
#include<iostream.h>
void main()
 {float x[100],aux;
  int n,i,j;

   // Citirea vectorului:
  cout<<"n=";  cin>>n;
  i=0;
  while(i<n)
    { cout<<"x"<<i<<"="; cin>>x[i];
      i++;
    }

    // Inversarea elementelor:
  i=0; j=n-1;
  while(i<j)
    {aux=x[i];  x[i]=x[j];  x[j]=aux;
     i++;   j--;
    }

   // Afisarea vectorului:
  cout<<" vectorul inversat este:";
  i=0;
  while(i<n)
    { cout<<x[i]<<" ";
      i++;
    }
```

```
}
```

2) Să se calculeze valoarea numărului π utilizând formula lui Madhava din Sangamagrama(anul ≈1400) $\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{\left(-\frac{1}{3}\right)^k}{2k+1}$

```
#include<iostream.h>
#include<iomanip.h>
#include <math.h> // pentru fabsl =valoarea absoluta
long double Pi()
 { long double eps=1.0e-20;
   long double pi=1.0, pia=0,t=1;
        /*  pi=valoarea la pasul curent, k;
            pia=valoarea de la pasul anterior;
            t=(-1/3)^k ;
        */
   unsigned long k=1;
   while(fabsl(pi-pia)>eps)
     { pia=pi;
       t*=-1.0/3.0;
       pi+=t/(2*k+1);
       k++;
     }
   // cout<<k<<endl;
   return sqrt(12)*pi;
 }

 void main()
  { cout<<endl<<setw(22)<<setprecision(20)<<Pi()<<endl;}
```

I.4.5. Instrucţiunea for

Instrucţiunea `for`, ca şi instrucţiunea `while` se utilizează pentru implementarea structurii repetitive cu test iniţial. Uzual, instrucţiunea for se foloseşte pentru implementarea ciclui cu contor.

Formatul ei este:

```
for( <exp1>; <exp2>; <exp3> )
   <instrucţiune>
```

unde

<exp1> se numeşte expresie de iniţializare şi se evaluează o singură dată, înaintea primei iteraţii, realizând iniţializarea ciclului. Uzual, ea atribuie o valoare iniţială variabilei de control a ciclului. Valoarea sau expresia de iniţializare poate lipsi in situaţia în care iniţializarea ciclului este făcută în afara sa, sau poate conţine mai multe expresii de iniţializare separate prin operatorul virgulă.

<exp2> se numeşte expresie de testare şi se execută înaintea fiecărei iteraţii, reprezentând condiţia de continuare aciclului. Ciclul se termină când această

expresie devine falsă. Dacă <*exp₂*> lipseşte, se consideră că expresia de test este adevărată tot timpul, iar ciclul se execută fără întrerupere.

<*exp₃*> specifică reiniţializările ce se efectuează după fiecare iteraţie; <instrucţiune> formează corpul ciclului, care se execută repetat.

Expresiile <exp1>,<exp2>,<exp3> pot fi şi vide. Totuşi caracterele ";" vor fi totdeauna prezente.

Instrucţiunea for este echivalentă cu:

```
<exp1>;
while(<exp2>)
  { <instrucţiune>
    <exp3>;
  }
```

Reciproc, orice instrucţiune while:

```
while(<exp>) <instrucţiune>
```

este echivalentă cu

```
for( ; <exp> ; ) <instrucţiune>
```

Instrucţiunea:

```
for(;;) <instrucţiune>
```

defineşte un ciclu "infinit" din care se iese prin alte mijloace decât cele obişnuite.

Exemple de utilizare a instrucţiunii for:

Programul următor calculează suma $S=\sum_{k=0}^{n}$ ¿¿¿

```
#include<iostream.h>
void main()
{ long double T,S;
  float a,x;
  int n,k;
  cout<<"n="; cin>>n;
  cout<<"a="; cin>>a;
  cout<<"x="; cin>>x;
  T=1;
  S=1;
  for(k=1;k<=n;k++)
   { T*=-a*x/k;
     S+=T;
   }
  cout<<"S="<<S<<endl;
}
```

2) Fie x_1, x_2,..., x_n ,numere întregi ce se introduc de la tastatură. Să se determine suma numerelor pozitive şi suma pătratelor numerelor negative.

```
#include<iostream.h>
void main()
{ int n,x,S1,S2,i;
  S1=0;
  S2=0;
  cout<<"n="; cin>>n;
  for(i=1;i<=n;i++)
    { cout<<"x"<<i<<"="; cin>>x;
      if(x>=0)S1+=x;
      else S2+=x*x;
    }
  cout<<"Suma numerelor pozitive="<<S1<<endl;
  cout<<"Suma patratelor numerelor negative="<<S2<<endl;
}
```

3) Fie a_0, a_1,..., a_{n-1} şi x_0, x_1,..., x_{n-1}, numere reale ce se introduc de la tastatură în vectorii a şi x. Programul următor calculează suma $S=\sum_{i=0}^{n-1} a_i \cdot x_i$

```
#include<iostream.h>

void main()
 {float a[100],x[100],S;
  int n,i;
  cout<<"n=";  cin>>n;
  for(i=0;i<n;i++)  { cout<<"a"<<i<<"="; cin>>a[i]; }
  for(i=0;i<n;i++)  { cout<<"x"<<i<<"="; cin>>x[i]; }

  S=0;
  for(i=0;i<n;i++)
       S=S+a[i]*x[i];
  cout<<"S="<<S;
 }
```

Putem avea mai multe cicluri for consecutive, ca de exemplu:

```
p=1;
for(i=1; i<=m; i++)
  for(j=1; j<=n; j++)
    for(k=1; k<=p; k++)
      p*=i+j+k;
```

I.4.6. Instrucţiunea do while

Această instrucţiune are formatul:

```
do
  <instrucţiune>
while(<expresie>);
```

şi implementează structura repetitivă cu test final.

Efect: Execută în mod repetat <*instrucţiune*> (simplă sau compusă) cât timp <*expresie*> este adevărată (diferită de zero).

Exemplu de utilizare a instrucţiunii do while:

Programul următor calculează suma unor produse de perechi de numere introduse de la tastatură cât timp suma rezultată este mai mica decât 1000.

```
#include<iostream.h>
void main()
 {long x,y,S;
  S=0;
  do
   { cout<<"x,y=";
     cin>>x>>y;      //se vor  tasta două numere întregi separate prin spaţiu
     S+=x*y;
   }
  while (S<1000);
  cout<<"S="<<S<<endl;
 }
```

I.4.7. Instrucţiunea break

Formatul acestei instrucţiuni este:

```
break;
```

Ea produce ieşirea forţată din instrucţiunile repetitive *while, do while* şi *for* sau dintr-o instrucţiune *switch*. Instrucţiunea *break* permite ieşirea dintr-un singur ciclu, nu şi din eventualele cicluri care ar conţine ciclul în care s-a executat instrucţiunea *break*.

Un exemplu de utilizare frecventă, il constituie ieşirea dintr-un ciclu infinit de forma:

```
for(;;)
 { ...
   if(...) break;
   ...
 }
```

Instrucţiunea *break* provoacă eroare dacă apare în afara instrucţiunilor *while, for, do while şi switch*.

Exemple de utilizare a instrucţiunii *break:*

1) Următorul program implementează jocul „Ghiceşte numărul!”.

```
#include<iostream.h>
#include<conio.h>
void main()
 {float a, b;
  cout<<"Propuneti un numar: ";cin>>a; // nr. ce trebuie ghicit
  clrscr(); // stergerea ecranului
  cout<<"Ghiciti numarul: ";  cin>>b;
  while(1) // ciclu infinit
   { if(a==b)  {cout<<"Ai ghicit nr!"; break;}
     if(a<b)  cout<<"Numar prea mare"<<endl;
     else     cout<<"Numar prea mic"<<endl;
      cout<<"Incercati alt numar:";  cin>>b;
   }
  getch();
 }
```

2) Următorul program afişează poziţia pe care se află un număr x în vectorul neordonat $(a_0, a_1,..., a_{n-1})$.

```
#include<iostream.h>
void main()
 {long a[100],x,poz;
  int n,i;
   // Citirea vectorului:
  cout<<"n=";
  cin>>n;
  cout<<"Tastati elementele vectorului:";
  for(i=0;i<n;i++) cin>>a[i];

    // Citirea lui x:
  cout<<"x=";
  cin>>x;

    // Determinarea pozitiei lui x in vector:
  poz=-1;
  for(i=0;i<n;i++) if(x==a[i]){poz=i; break;}
  if(poz>=0) cout<<x<<"se afla in vector pe pozitia "
                 <<poz<<endl;
  else    cout<<x<<"nu se afla in vector "<<endl;
 }
```

3) Următorul program verifică dacă un vector format din n elemente numere reale, este ordonat crescător.

In program se foloseşte variabila semafor pentru a indica dacă vectorul este ordonat crescător (semafor==1) sau nu (semafor==0).

```
#include<iostream.h>
void main()
 {float a[100];
  int n,i;

  cout<<"n=";  cin>>n;
  cout<<"Tastati "<<n<<"numere separate prin spatiu:"<<endl;
  for(i=0;i<n;i++)cin>>a[i]; //citirea elementelor vectorului

  int semafor;
  semafor=1;

  for(i=1;i<n;i++)
   if(a[i]<a[i-1]){semafor=0;break; /*vectorul nu este ordonat crescator*/ }
  if(semafor==1)
   cout<<"vectorul este ordonat crescator"<<endl;
  else
   cout<<"vectorul nu este ordonat crescator"<<endl;
 }
```

I.4.8. Instrucţiunea continue

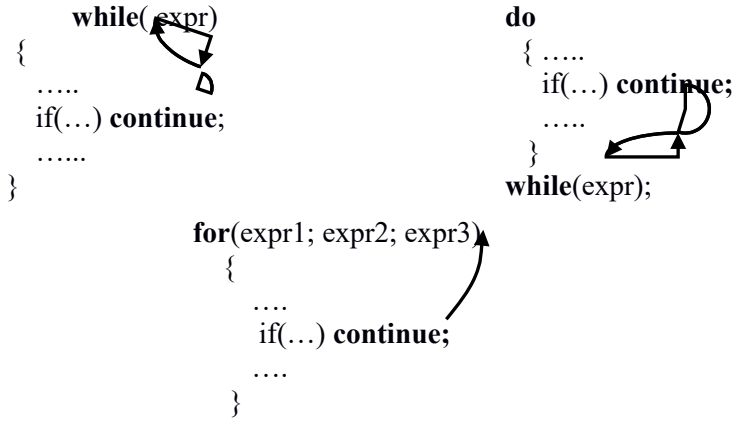
Are formatul:

```
continue;
```

Se utilizează în corpul unui ciclu şi are următorul efect:

• în ciclurile *while* şi *do while* ea realizează saltul la evaluarea expresiei care decide asupra continuării ciclului;

• în ciclul *for* ea realizează saltul la pasul de reiniţializare.



Astfel, programul

```
#include<iostream.h>
void main()
 { int i;
   for (i = 0; i < 5; i++)
    { if (i == 3) continue;
      printf("i = %d\n",i);
    }
   cout<<"valoarea lui i la iesirea din ciclul for este "<<i;
}
```

Va afisa urmatoarele rezultate

i = 0
i = 1
i = 2
i = 4
valoarea lui i la iesirea din ciclul for este 5

Exemplu de utilizare a instrucţiunilor continue şi break:

Se introduc de la tastatură numere întregi (pozitive şi negative) cît timp suma numerelor pozitive este mai mică decât 1000. Programul afişează pătratul numerelor pozitive.

```
#include<iostream.h>
#include<conio.h>
void main()
 { long S,x;
   S=0;
   for(;;)  // ciclu infinit
    { cout<<"x=";  cin>>x;
        if(x<=0) continue; /* salt la partea de reiniţializare a
instrucţiunii for
                                        (în cazul de faţă  expresia de
reiniţializare este vidă)*/
      S+=x;
      if(S>=1000) break;      // ieşire din for
      cout<<"x*x="<<x*x<<endl;
    }
  }
```

I.4.9. Instrucţiunea switch

Implementează structura de selecţie multiplă. Sintaxa acestei instrucţiuni este:

```
switch(<expresie>)
  {  case <c1>: <sir1>
     case <c2>: <sir2>
     ...
     case <cn>: <sirn>
     default  : <sir>
```

```
  }
```

unde

<*c*$_1$>,<*c*$_2$>,...,<*c*$_n$> sunt expresii constante de tip întreg,

<*expresie*> este o expresie de tip întreg (orice tip întreg),

<*sir*$_1$>,...,<*sir*$_n$> sunt şiruri de instrucţiuni (un astfel de şir poate fi şi vid).

Efect:

1) Se evaluează <*expresie*>.

2) Se compară valoarea expresiei <*expresie*>, succesiv, cu valorile <c$_1$>,<c$_2$>,...,<c$_n$>.

3) Dacă valoarea <*expresie*> coincide cu <c$_k$>, se execută secvenţa de instrucţiuni <sir$_k$>, <sir$_{k+1}$>,...,<sir$_n$>,<sir>.

Dacă în această secvenţă se întâlneşte instrucţiunea `break`, atunci aceasta are ca efect ieşirea din instrucţiunea `switch`.

4) În cazul în care valoarea expresiei nu coincide cu niciuna din constantele <c$_1$>,<c$_2$>,...,<c$_n$> se execută secvenţa de instrucţiuni definită de <*sir*>. Alternativa `default` nu este obligatorie, în lipsa ei, dacă valoarea <*expresie*> nu coincide cu niciuna din constantele <c$_1$>,<c$_2$>,...,<c$_n$> instrucţiunea `switch` nu are niciun efect.

Exemplu de utilizare a instrucţiunii `switch`:

Programul următor calculează funcţia

$$f(x,k)=\begin{cases}\sin x, & k=1\\ \cos x, & k=2\\ \operatorname{tg} x, & k=3\\ \operatorname{ctg} x, & k=4\end{cases}$$

¿
¿

```
#include<math.h>
#include<conio.h>
#include<iostream.h>

void main()
 { int k;
   float x,f;
   float r; // pentru conversia lui x în radiani
   cout<<"k="; cin>>k;
   cout<<"x="; cin>>x;
   r=x*3.141592/180;
   switch(k)
     { case 1: f=sin(r);
               break;
       case 2: f=cos(r);
               break;
       case 3: f=tan(r);
               break;
       case 4: f=1/tan(r);
               break;
       default: f=sqrt(5+2*sin(r)+3*cos(r));
      }
    cout<<"f("<<k<<","<<x<<")="<<f<<endl;
```

```
  }
```

I.4.10. Instrucţiunea vidă

Pentru instrucţiunea vidă nu există cuvânt rezervat, prezenţa ei este marcată prin caracterul punct şi virgulă şi nu are niciun efect asupra variabilelor, starea acestora rămânând neschimbată; Instrucţiunea vidă este necesară în anumite situaţii de programare. De exemplu, poate fi utilă pentru un ciclu fără instrucţiuni în corpul său:

```
i=0;
while(x[i++]=y[i]); /* copiază elementele vectorului y în vectorul x până la
                                        întâlnirea primului element zero din y
*/
```

I.4.11. Instrucţiunea goto

Prin etichetă înţelegem un nume urmat de două puncte (:)

<nume>:

Etichetele sunt locale funcţiei şi prefixează instrucţiuni. Instrucţiunea goto are formatul

```
goto <nume>;
```

Ea realizează saltul la instrucţiunea prefixată de *<nume>*:

Se recomandă folosirea instrucţiunii goto când dorim să ieşim dintr-un ciclu inclus în mai multe cicluri.

I.4.12. Apelul unei funcţii

O funcţie de forma:

```
void <nume funcţie>(<lista parametrilor formali>)
     { ... }
```

care nu produce o valoare directă, se apelează printr-o instrucţiune de apel cu următorul format:

```
<nume funcţie>(<lista parametrilor efectivi>);
```

O funcţie de forma

```
<tip returnat> <nume funcţie>(<lista parametrilor formali>)
     { ... }
```

unde `<tip returnat>` este diferit de `void` (prin urmare returnează valori directe), poate fi apelată fie printr-o instrucţiune de apel, când nu dorim să utilizăm valoarea returnată, fie sub forma unui operand al unei expresii când utilizăm valoarea returnată. In exemplul următor funcţia `int getch(void)` (returnează codul ASCII al caracterului citit de la tastatură), este apelată în ambele variante.

```
#include<conio.h>
#include<iostream.h>
void main()
```

```
 {char c;
  cout<<"tastati un caracter";
  c=getch();     //citeşte caracterul tastat şi îl memorează în c
(folosim valoarea returnată)
  cout<<c<<endl;
  getch();          // citeşte caracterul tastat fără memorare  (nu
folosim valoarea returnată)
 }
```

I.4.13. Instrucţiunea return

Revenirea dintr-o funcţie se poate face în două moduri:

• la întâlnirea instrucţiunii `return`;

• după execuţia ultimei sale instrucţiuni, adică a instrucţiunii ce precede acolada închisă ce termină corpul funcţiei respective. În această situaţie funcţia nu returnează nicio valoare.

Instrucţiunea `return` are două formate:

```
return;
```

caz în care funcţia nu returnează un rezultat direct(tipul funcţiei este void), sau

```
return <expresie>
```

caz în care funcţia returnează valoarea expresiei <*expresie*> (convertită, dacă este cazul, la tipul funcţiei).

Exemplu:

In următorul program funcţia `int prim(long n)` returnează 1 dacă argumentul este număr prim şi 0 în caz contrar.

```
#include<iostream.h>
#include<math.h> //pentru funcţia sqrt() - radical de ordin 2

int prim(long n) // verifica daca n ≥1 este prim
 { if(n==1) return 0; //1 nu este prim
   if(n==2||n==3) return 1; // 2 si 3 sunt nr. prime
   if(n%2==0) return 0; // n nu este prim pentru ca se divide cu 2
   long r=sqrt(n);
   for( long d=3; d<=r; d+=2)
     if(n%d==0) return 0;
   return 1;
 }
void main()
  { long n;
    cout<<"n="; cin>>n;
    if(prim(n))cout<<n<<" este numar prim "<<endl;
    else       cout<<n<<" nu este numar prim "<<endl;
  }
```

I.5. Sfera de influenţă a variabilelor

În funcţie de locul de declarare, variabilele pot fi: globale când sunt declarate în afara funcţiilor şi locale când sunt declarate in interiorul funcţiilor.

Variabilele globale formează nivelul extern. Aceste variabile sunt accesibile din orice funcţie a fişierului sursă care urmează declaraţiei. Variabilele globale se alocă la compilare şi rămân în memoria calculatorului pe tot parcursul executării programului, de aceea se mai numesc şi permanente.

Variabilele locale formează nivelul intern si este format din declaraţiile conţinute în interiorul blocurilor formează (prin bloc înţelegem o instrucţiune compusă sau corpul unei funcţii, adică o succesiune de instrucţiuni delimitate de acolade). Aceste variabile pot fi folosite (sunt vizibile) doar în blocul în care au fost declarate sau într-un bloc subordonat acestuia. La nivel inferior putem declara o variabilă având aceeaşi denumire cu a uneia declarate la nivel superior. In acest caz noua declaraţie va fi valabilă la acest nivel şi la nivelele inferioare (subordonate), iar în nivelele superioare rămâne valabilă declaraţia iniţială.

Variabilele locale pot fi declarate statice (prin utilizarea cuvântului cheie „*static*”), urmând să fie alocate la compilare şisă ramână în memoria calculatorului pe tot parcursul executării programului. Variabilele locale nestatice sunt create şi li se alocă spaţiu în memoria calculatorului numai în momentul în care se execută blocul de program în care este declarată variabila. La încheierea execuţiei blocului respectiv, variabila dispare şi spaţiul de memorie va fi alocat altor blocuri. Dacă se revine ulterior în blocul iniţial, variabila va fi realocată şi poate să primească altă adresă. Variabilele de acest fel se numesc variabile cu alocare automată a adresei (variabile *automatice*). Alocarea variabilelor din clasa "*auto*" se face pe stiva sistemului. Variabilele locale, al căror domeniu de valabilitate se limitează la un bloc sunt în mod implicit variabile automatice, chiar dacă nu se menţionează în mod explicit acest lucru. De asemenea, parametrii formali sunt variabile din clasa "auto" şi deci se alocă pe stiva sistemului. Dacă dorim ca o variabilă locală să nu fie alocată pe stivă, deci să nu fie "auto", o declarăm obişnuit, dar declaraţia va fi precedată de cuvântul "static":

```
static  <tip>  <lista de nume>;
```

Exemplu:

```
float f()
  { int k;
    static int a[5];
   // ...
  }
```

Variabila simplă `k`, precum şi variabila tablou `a` sunt cunoscute şi pot fi referite în interiorul blocului în care au fost declarate. Se mai spune că ele sunt locale. Variabila tablou `a`, descrisă prin cuvântul cheie static, îşi păstrează aceeaşi adresă pe toată durata programului, adresă pe care o primeşte la începutul executării programului. Variabilei `k` i se alocă spaţiu pe stivă de fiecare dată când se execută funcţia `f()`, la adrese care pot fi diferite.

- Declaraţia informativă "extern"

Pentru ca o variabilă globală să poată fi folosită de funcţii situate în alt fişier sursă, sau în cadrul aceluiaşi fişier în funcţii anterioare declarării variabilei, trebuie ca acea variabilă să fie descrisă printr-o declaraţie "*extern*" în funcţiile respective sau în afara oricărei funcţii ale noului fişier. Acum fişierele pot fi compilate separat şi linkeditate împreună.

Exemplu:

Fişierul F1.CPP

```
...
int x,y;
...
```

Fişierul F2.CPP

```
...
extern int x,y;
int suma(){return x+y;}
```

Fişierele se pot compila separat.

Pentru obţinerea executabilului vom include în fişierul F2.CPP, fişierul F1.CPP cu ajutorul directivei #include"F1.CPP" sau vom crea un *proiect*. Pentru crearea proiectului vom selecta, din meniul turbo C, `Open project`. Alegem pentru proiect, de exemplu, numele PR1. Includem fişierele F1.CPP şi F2.CPP utilizând tasta funcţională <*Insert*>. Compilarea proiectului se face cu `Build all` din `Compile`. Ştergerea unui fişier din proiect se face prin poziţionarea pe fişierul respectiv cu ajutorul săgeţilor şi apoi acţionarea tastei `<Delete>`.

Exemplificare:

Program pentru ordonarea crescătoare a unui vector de numere întregi:

```
#include<iostream.h>
#include<conio.h>

long x[100];
int n;      //  variabile globale

void citeste()    // funcţie pentru citirea vectorului
 { int i; //variabilă locală
   cout<<"n=";   cin>>n;
   cout<< "Tastati "<<n<<" numere intregi:"
   for(i=0;i<n;i++)
     { cout<<"x["<<i+1<<"]="; cin>>x[i];}
 }

void sorteaza()    // funcţie pentru ordonarea vectorului
 { int i,j; //variabile locale
   for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
```

```
     if(x[i]>x[j])
      { long aux; //variabilă locală
        aux=x[i]; x[i]=x[j]; x[j]=aux;
      }
 }
void afiseaza()   // funcţie pentru afişarea vectorului
  { int i;
    for(i=0;i<n;i++)cout<<x[i]<<" ";
  }
void main()      //  funcţia principală
  {  clrscr();
     citste();
     cout<<"X=";
     afiseaza();  cout<<endl;
     sorteaza();
     cout<<"Vectorul sortat: "<<endl;
     afiseaza(); cout<<endl;
    getch();
   }
```

I.6. Iniţializarea variabilelor

Iniţializarea variabilelor simple

O variabilă simplă se poate iniţializa printr-o declaraţie de forma:

```
<tip> <nume> = <expresie>;
```

sau

```
static <tip> <nume> = <expresie>;
```

dacă variabila este statică.

În cazul variabilelor globale şi statice, expresia utilizată trebuie să fie o expresie constantă, care să poată fi evaluată de compilator la întâlnirea ei. Aceasta, deoarece variabilele globale şi statice se iniţializează prin valori definite la compilare. Variabilele automatice se iniţializează la execuţie, de fiecare dată când se activează funcţia în care sunt declarate. Din această cauză, expresia utilizată la iniţializare nu mai este necesar să fie o expresie constantă. Variabilele automatice care nu sunt iniţializate vor conţine valori întâmplătoare şi, deoarece ele apar şi dispar odată cu blocul în care au fost declarate, nu îşi păstrează valorile de la o execuţie la alta a blocului din care aparţin. Spre deosebire de variabilele automatice, variabilele globale şi statice primesc în mod automat valoarea zero la începutul executării programului, dacă nu sunt iniţializate în mod explicit. Variabilele statice sunt iniţializate numai o singură dată, la începutul executării programului, şi îşi păstrează valorile de la o execuţie la alta a blocului în care au fost declarate.

Exemplul 1:

```
#include<iostream.h>
void incrementare()
  { int i=1;
    static int k=1;
```

```
    i++;
    k++;
    cout<<"i="<<i<<", k="<<k<<endl;
  }
void main()
  { incrementare();
    incrementare();
    incrementare();
  }
```

Rezultatul execuţiei va fi:

```
i=2, k=2
i=2, k=3
i=2, k=4
```

Variabilele interne statice oferă posibilitatea păstrării în permanenţă a unor informaţii ce aparţin funcţiei. O astfel de variabilă ar putea fi folosită pentru a memora de câte ori a fost apelată o funcţie.

Exemplul 2:

```
#include <iostream.h>
void f1()
  { static int k; // implicit iniţializată cu 0
    k++;
    cout<<"f1 apelul"<<k<<endl;
  }

void f2()
  { static int k; /* Variabilă iniţializată implicit cu 0.  Este variabilă internă funcţiei f2(),
                     cu alocare statică la o adresă diferită de adresa variabilei k, internă funcţiei f1.*/
    k++;
    cout<<"f2 apelul"<<k<<endl;
  }

void main()
  { f1(); f1(); f2(); f1(); f2(); }
```

Rezultatul execuţiei este:

```
f1 apelul 1
f1 apelul 2
f2 apelul 1
f1 apelul 3
f2 apelul 2
```

Iniţializarea variabilelor tablou

Un tablou unidimensional se poate iniţializa folosind formatul:

<*tip*> <*nume*> [<*dim*>]={<e_1>,<e_2>,...,<e_n>};

sau

```
static <tip> <nume> [<dim>]={<e1>,<e2>,...,<en>};
```

Numărul expresiilor <e_i> de iniţializare poate fi mai mic decât al numărului elementelor tabloului. Elementele neiniţializate au valoarea iniţială 0.

În cazul în care se iniţializează fiecare element al tabloului, numărul <*dim*> al elementelor acestuia nu mai este obligatoriu în declaraţia tabloului respectiv. Deci putem scrie:

```
<tip> <nume> []={<e1>,<e2>,...,<en>}
```

Numărul elementelor tabloului fiind considerat egal cu numărul expresiilor.

Pentru un tablou bidimensional vom folosi următorul format:

```
<tip><nume>[<dim1>][<dim2>]={{<e11>,<e12>,...,<e1 m1>},
                             {<e21>,<e21>,...,<e2 m2>},
                                        ...
                             {<en1>,<en1>,...,<en mn>}
                            };
```

sau

```
static <tip> <nume> [<dim1>][<dim2>]={{<e11>,<e12>,...,<e1 m1>},
                                      {<e21>,<e21>,...,<e2 m2>},
                                                 ...
                                      {<en1>,<en1>,...,<en mn>}
                                     };
```

Numerele m_1, m_2,..., m_n, pot fi mai mici decât <*dim*$_2$> în oricare din acoladele corespunzătoare ale tabloului, de asemenea *n* poate fi mai mic decât <*dim*$_1$>. În aceste situaţii restul elementelor tabloului vor fi iniţializate cu 0. Dacă n este egal cu <*dim*$_1$>, atunci <*dim*$_1$> poate fi omis, dar <*dim*$_2$> este obligatoriu.

Într-o modalitate asemănătoare se pot iniţializa şi tablouri cu mai multe dimensiuni.

Exemplu

```
int  a[2][3]={ {1,2,3},
               {4,5,6}
             };
```

Un tablou multidimensional se poate iniţializa şi astfel:

```
int t[2][3]={1,2,3,4,5,6};
```

sau

```
int t[][3]={1,2,3,4,5,6};
```

Tablourile de tip caracter pot fi iniţializate astfel:

```
char <nume>[<dim>]=<sir de caractere>;
```

sau

```
static char <nume>[<dim>]=<sir de caractere>;
```

Compilatorul adaugă automat caracterul NULL (\0) după ultimul caracter al şirului utilizat în iniţializare. Numărul <*dim*> poate fi omis.

Deci declaraţia:

```
char t[4]={'a','b','c','\0'};
```

este echivalentă cu:

```
char t[4]="abc";
```

şi cu:

```
char t[]="abc";
```

I.7. Transferul parametrilor la apelul funcţiilor

La apelul unei funcţii, fiecărui parametru formal îi corespunde un parametru efectiv. In C++ sunt implementate două metode de transmitere a parametrilor la funcţii:

prin valoare – când o eventuală modificare a parametrului în funcţie nu afectează valoarea parametrului efectiv în funcţia apelantă. În cazul apelului prin valoare, se transferă funcţiei apelate valoarea parametrului efectiv care poate fi o constantă, o variabilă sau o expresie.

prin referinţă – în care variabila transmisă funcţiei ca parametru efectiv este afectată de eventualele modificări aduse în funcţie. Pentru aceasta funcţia apelată trebuie să dispună de adresa parametrului efectiv pentru ca să-l poată modifica.

Sunt trei modalităţi de a realiza transferul prin referinţă:

- prin utilizarea ca parametru a numelui unui tablou ;
- prin utilizarea ca parametru a unei variabile de tip pointer ;
- prin utilizarea ca parametru a unei variabile de tip referinţă.

Numele unui tablou are ca valoare chiar adresa primului său element, în consecinţă, dacă un parametru formal este numele unui tablou atunci la apel se va transmite funcţiei adresa de început a tabloului ce se utilizează efectiv, şi prin urmare îi pot fi modificate elementele.

Deoarece compilatorul nu foloseşte dimensiunea tabloului transmis ca parametru, ci doar adresa lui de început, pentru parametrul formal de tip tablou putem folosi sintaxa de declarare următoare:

```
<tip> <nume tablou>[]
```

I.7.1. Probleme rezolvate.

1) Operaţii cu matrice

Citirea si afişarea matricelor, suma şi produsul a două matrice

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>

void citire(int a[10][10],int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      { cout<<"Linia "<<i+1<<":";
         // tastaţi n numere separate prin spaţiu
        for(j=0;j<n;j++) cin>>a[i][j];
      }
  }
```

```
void suma(int a[10][10],int b[10][10],int c[10][10],
          int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        c[i][j]=a[i][j]+b[i][j];
  }

void produs(int a[10][10],int b[10][10],int c[10][10],
            int m,int n,int p)
     // a cu m linii si n coloane,  b cu n linii si p coloane,  c cu m linii si p
coloane
  { int i,j,k,s;
    for(i=0;i<m;i++)
      for(j=0;j<p;j++)
       { s=0;
         for(k=0;k<n;k++)
           s+=a[i][k]*b[k][j];
         c[i][j]=s;
       }
  }

void afisare(int a[10][10],int m,int n)
  { int i,j;
    for(i=0;i<m;i++)
      { for(j=0;j<n;j++)
         cout<<setw(5)<<a[i][j];
        cout<<endl;
      }
  }

void main()
  { clrscr();
    int x[10][10],y[10][10],z[10][10];
    int n;
    cout<<"n="; cin>>n;   // n linii, n coloane
    cout<<"Matricea X:"<<endl;
    citire(x,n,n);
    cout<<"Matricea Y:"<<endl;
    citire(y,n,n);
    suma(x,y,z,n,n);
    cout<<"Z=X+Y:"<<endl;
    afisare(z,n,n);
    produs(x,y,z,n,n,n);
    cout<<"Z=X*Y:"<<endl;
    afisare(z,n,n);
    getch();
  }
```

2) Parcurgere în spirală

*Fie a o matrice cu n linii şi n coloane. Să se construiască vectorul b cu n*n elemente, obţinute prin parcurgerea matricei în spirală, din colţul din stânga-sus către dreapta, până în centrul matricei.*

```
#include<iostream.h>
#include<iomanip.h> //pentru setw

void citireMatrice(int a[10][10], int n)
 { int i,j;
   for(i=0;i<n;i++)
    { cout<<"linia "<<i<<":";
      for(j=0;j<n;j++)
       cin>>a[i][j];
    }
 }
void afisareMatrice(int a[10][10],int n)
 { int i,j;
   for(i=0;i<n;i++)
    { for(j=0;j<n;j++)
       cout<<setw(6)<<a[i][j];
     cout<<endl;
    }
 }
void parcurgereInSpirala(int a[10][10], int n,
                         int b[], int &nn )
 { int k,p,q,r,i,j;
   nn=n*n;
   k=0;
   p=0; q=n-1; r=n-1; //p,q,r definesc conturul de parcurs (p,p)->(p,q)->(q,r)->(r,p-1)
   while(k<n*n)
    { for(j=p;j<=r;j++)b[k++]=a[p][j];   //parcurgere stanga->dreapta
      for(i=p+1;i<=q;i++)b[k++]=a[i][r]; //sus ->jos (col. din dreapta)
      for(j=r-1;j>=p;j--)b[k++]=a[q][j]; // dreapta->stanga(linia de jos)
      for(i=q-1;i>=p+1;i--)b[k++]=a[i][p]; // jos->sus (col din stanga)
      p++;q--;r--; //urmatorul contur
    }
 }
void afisareVector(int b[],int n)
 { for(int i=0;i<n;i++)cout<<b[i]<<" "; }
int main()
 {int a[10][10],n;
  int b[100],nn;
  cout<<"nr. linii(coloane)="; cin >>n;


  citireMatrice(a,n);
  afisareMatrice(a,n);
  parcurgereInSpirala(a, n, b, nn);
```

```
  cout<<"Vectorul obtinut prin parcurgerea matricei in
spirala:"<<endl;
  afisareVector(b,nn);
 }
```

3) Problema celor 4 triunghiuri

O matrice pătratică este împărţită de cele două diagonale în patru triunghiuri. Să se determine suma elementelor din cele patru triunghiuri. Elementele de pe diagonale fac parte din triunghiurile respective.

```
#include<iostream.h>
long sumaTrSus(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[p][k];
       p++;q--;
     }
    return S;
  }
long sumaTrJos(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[q][k];
       p++;q--;
     }
    return S;
  }
long sumaTrStanga(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
         S+=a[k][p];
       p++;q--;
     }
    return S;
  }
long sumaTrDreapta(long a[20][20],int n)
  { long S=0;
    int p,q,k;
    p=0;q=n-1;
    while(p<=q)
     { for(k=p;k<=q;k++)
```

```
        S+=a[k][q];
      p++;q--;
    }
   return S;
  }
void main()
  {long a[20][20]={{1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5},
                   {1,2,3,4,5}};
   cout<<sumaTrSus(a,5)<<endl;
   cout<<sumaTrStanga(a,5)<<endl;
   cout<<sumaTrJos(a,5)<<endl;
   cout<<sumaTrDreapta(a,5)<<endl;
  }
```

4) Căutare binară

Algoritmul de căutare binară este un algoritm de căutare folosit pentru a găsi un element într-un vector ordonat. Fie a un vector ordonat crescător şi x un element ce se caută în vectorul a. Valoarea x este comparată cu valoarea elementului din mijlocul vectorului a. Dacă cele două valori sunt egale, algoritmul se termină. Dacă valoarea lui x este mai mică decât acea valoare, căutarea se efectuează, prin acelaşi procedeu, pentru elementele de la începutul vectorului până la mijloc, iar dacă este mai mare, căutarea se efectuează de la mijlocul vectorului până la sfârşitul său. Întrucât la fiecare pas cardinalul mulţimii de elemente în care se efectuează căutarea se înjumătăţeşte, algoritmul are complexitate logaritmică.

```
#include<iostream.h>
#include<stdlib.h>

int caut(long a[],int n,long x)
 { int i=0,j=n-1,m;
   while(i<=j)
     { m=(i+j)/2;
       if(a[m]==x) return m;// x se gaseste pe pozitia m
       if(x<a[m])j=m-1;
       else i=m+1;
     }
   return -1; // x nu se gaseste in vectorul a
 }

void main()
{
 long a[100], x;
 int n, poz;
 cout<<"n=";
 cin>>n;
```

```
cout<<"Tastati "<<n<<" elemente in ordine crescatoare, "
     <<"separate prin spatiu:"<<endl;
cin>>a[0];

for(int i=1;i<n;i++)
 {cin>>a[i];
  if(a[i]<a[i-1])
   { cout<<"vectorul nu este ordonat crescator"<<endl;
     return;
   }
 }
while(1)
 {cout<<"Tastati valoarea lui x "
      <<" (sau CTRL+Z pentru sfarsit):";
  cin>>x;  if(cin.eof()) break;
  poz=caut(a,n,x);
  if(poz>=0)
   cout<<x<<" se gaseste in vectorul a pe pozitia "
       <<poz<<endl;
   else
    cout<<x<<" nu se gaseste in vectorul a"<<endl;
 }
```

5) Exemple de funcţii de lucru cu şiruri de caractere:

Determinarea numărului de caractere dintr-un şir de caractere:

```
int lungime(char s[])
    { int k=0;
      while(s[k]) k++;
      return k;
    }
```

Copierea unui şir de caractere:

```
void copiaza(char sursa[], char dest[])
    { int i=0;
      while(dest[i++]=sursa[i]);
    }
```

Observaţie: Biblioteca <*string.h*> conţine funcţii pentru determinarea lungimii şi pentru copierea şirurilor(*strlen* respectiv *strcpy*).

Inversarea caracterelor unui şir de caractere:

```
void inversare(char s[])
  { int i,j;
    char c;
    for(i=0,j=lungime(s)-1; i<j; i++,j--)
      { c=s[i]; s[i]=s[j]; s[j]=c; }
```

```
  }
```

Determinarea primei poziţii de unde începe un subşir într-un şir de caractere.

```
int cauta(char sir[], char subsir[])
  { int k,j,lgSubsir,pozMax;
    lgSubsir=lungime(subsir);
    pozMax=lungime(sir)-lgSubsir;
    for(k=0; k<= pozMax; k++)
      { for ( j=0; j<lgSubsir && subsir[j]==sir[k+j]; j++ );
          if(j==lgSubsir) return k;
      }
    return -1;
  }

#include<iostream.h>

void main()
  { char a[25],b[25];
    cout<<"Sirul a=";  cin>>a;
    cout<<"Sirul b=";  cin>>b;
    cout<<"Sirul a are "<<lungime(a)<<" caractere."<<endl;
    cout<<"Sirul b are "<<lungime(b)<<" caractere."<<endl;
    cout<<"Subsirul b apare in a începând de la poz. "
          <<cauta(a,b)<<endl;
  }
```

6) Se introduce un text de la tastatură. Să se afişeze frecvenţa literelor mari şi a literelor mici din textul introdus.

```
#include  <stdio.h> //pentru gets si printf
#include <conio.h> //pentru  clrscr
#include  <string.h> //pentru strlen
#include  <ctype.h> // pentru islower si isupper

void main ( )
{ int i, n, a[26], b[26] ;
  char sir[1000],c ;

  clrscr( );

  printf(" Introduceti sirul:"); gets(sir);
  n=strlen(sir);

  for(i=0;i<26;i++)
    {a[i]=0; b[i]=0;}

  for(i=0;i<n;i++)
   {c=sir[i];
    if(islower(c))a[c-'a']++;
    if(isupper(c))b[c-'A']++;
```

```
   } //'a'=97, ...,'z'=122; 'A'=65, ..., 'Z'=90

  for(i=0;i<26;i++)
   { if(a[i]!=0)printf("Litera %c apare de %d ori \n",
                       (char)(i+97),a[i]);
     if(b[i]!=0)printf("Litera %c apare de %d ori \n",
                       (char)(i+65),b[i]);
   }
 }
```

7) Operaţii cu numere naturale mari: suma, diferenţa aritmetică, compararea şi produsul a două numere mari; împărţirea unui număr mare la un număr întreg. Algoritmi clasici.

Vom considera numerele reprezentate în baza 10. Cifrele numerelor mari, completate cu zerouri nesmnificative, vor fi memorate în vectori cu acelaşi număr de elemente. Dimensiunea comună a vectorilor este aleasă astfel încât să fie suficientă şi pentru memorarea numerelor rezultate in urma operaţiilor aplicate lor.

```
#include <iostream.h>
#include <conio.h>
#include<string.h>

int dim=100;   // dimensiunea comuna a vectorilor(nr maxim de cifre);
valoare implicita 100

int suma( int u[],int v[],int w[]) //w=u+v
 {int t=0,c; // t - cifra de transport
  for(int i=dim-1; i>=0;i--)
   { c=u[i]+v[i]+t;
     if(c>=10) { w[i]=c-10; t=1;} else {w[i]=c; t=0;}
   }
  return 1-t; // 1 adunare cu succes, 0 - depasire
 }
int diferenta(int u[],int v[], int w[]) // w=u-v, unde u>v
 {int t=0,c;
  for(int i=dim-1;i>=0;i--)
   { c=u[i]-v[i]+t;
     if(c<0){w[i]=10+c; t=-1;}
     else {w[i]=c; t=0;}
   }
  return 1+t;
 }

int comparare(int u[], int v[]) //1 daca u>v, 0 daca u=v, -1 daca
u<v
 {
  for (int i=0;i<dim;i++)
   {  if(u[i]<v[i])return -1;
```

```
      if (u[i]>v[i]) return 1;
   }
  return 0;
 }
```

/* Produsul a doua numere mari se calculeaza inmultind fiecare cifra a inmultitorului cu fiecare cifra a deinmultitului si adunind rezultatul la ordinul de marime corespunzător. */

```
int produs(int u[],int v[],int w[]) //w=u·v
{int i,j,t=0,s;
  for (i=0;i<dim;i++) w[i]=0;
  for( j=dim-1;j>=0;j--)
   for(i=dim-1;i>=0;i--)
    { int k=i+j+1-dim;
      s=u[i]*v[j]+t;
      if(k>=0)
       { w[k]+=s;
         t=w[k]/10;
         w[k]%=10;
       }
      else if(s!=0)return 0;// esec! dimensiune prea mica
    }
  return t==0?1:0;   //1 succes; 0 esec
 }

void impartire(int u[],long q,int w[],long &r) //w=u·q+r
 { int i;
   long p;
   r=0;
   for(i=0;i<dim;i++)
    {p=r*10+u[i];
     w[i]=p/q;
     r=p%q;
    }
 }

void afisare( int u[])
 { int i=0;
   while(i<dim-1 && u[i]==0)i++; // salt peste cifrele 0 nesemnificative
   while(i<dim){ cout<<u[i]; i++;}
}

int citire(int v[])
 { char s[255];
   cin>>s;
   int m=strlen(s);
   if(m>dim) return 0;  //esec; dimensiune insuficienta
   int i,j;
   for(i=m-1,j=dim-1; i>=0; i--,j--)
```

```
    {v[j]=s[i]-'0';
     if(v[j]<0||v[j]>9) return 0; //esec; caracter nenumeric
    }
  for( ;j>=0;j--)v[j]=0;
  return 1; //citire cu succes
}
void main()
{ int u[100], v[100],s[100],d[100],p[100],r[100];
 dim=100;
  cout<<"u=";
  citire(u);
  cout<<"v=";
  citire(v);

  suma(u,v,s);
  afisare(u); cout<<"+"; afisare(v);
  cout<<"="; afisare(w); cout<<endl;

  if(comparare(u,v)>0)diferenta(u,v,d);
  else diferenta(v,u,d);
  cout<<"|";afisare(u); cout<<"-"; afisare(v);
  cout<<"| ="; afisare(d); cout<<endl;

  int ok=produs(u,v,p);
  afisare(u); cout<<"*"; afisare(v);
  cout<<"="; afisare(p);
  if(!ok)cout<<"esec! dim prea  mica"; cout<<endl;

  long q,rest;
  cout<<"q="; cin>>q;
  impartire(u,q,r,rest);
  afisare(u); cout<<":"; cout<<q<<"="; afisare(r);
  cout<<" rest "<<rest<<endl;
}
```

8) Fiind date n numere întregi $a_1, a_2,..., a_n$ nu în mod necesar diferite, există totdeauna o submulţime a acestei mulţimi de numere, cu proprietatea că suma elementelor sale este divizibilă prin n. Următorul program determină o astfel de submulţime.

Soluţie.

Fie $s_k = a_1 + a_2 + ... + a_k$, $k = 1, ...$,n.

Dacă $(\exists) s_k$ a.î. $n \square s_k$ problema este rezolvată, altfel resturile sunt nenule `(∀) k∈{1,2,...,n}` şi aparţin mulţimii `{1,...,n-1}` $\Rightarrow$ `(∃)` două sume cu resturi egale. Fie s_k şi s_i, `i<k`, două asemenea sume $\Rightarrow$ $n \mid s_k - s_i$.

Dacă s_k `mod n = r`, vom memora indicele `k` în variabila `rest[r]`. În situaţia în care în `rest[r]` este deja memorat un indice nu ne rămâne decât să afişăm soluţia.

```
#include<iostream.h>
int a[101],rest[101],n;
void citeste()
  { cout<<"n="; cin>>n;
    cout<<"Tastati "<<n<<" numere:";
    int i;
    for(i=1;i<=n;i++){ cin>>a[i];rest[i]=-1;}
    rest[0]=0;
  }
void solutie()
  { int s,k,i,r;
    s=0;
    for(k=1;k<=n;k++)
     {s+=a[k];
      r=s%n;
      if(rest[r]==-1) rest[r]=k;
      else { cout<<"Solutie:";
             for(i=rest[r]+1; i<=k; i++)
                cout<<a[i]<<" ";
             cout<<endl;
             return;
            }
      }
  }

void main()
  { citeste();
    solutie();
  }
```

I.8. Capcane în programare

1) Ce rezultat furnizează programiul următor?

```
#include<iostream.h>
void main()
 { float x=1.0;
   x=x/10;
   cout<<”x*10=”<<x*10<<endl;
   if(x*10==1) cout<< " Adevarat! x*10 == 1 "<< endl;
   else        cout<< " Fals! x*10 != 1 "<< endl;
 }
```

Comentariu. Programul afişează un rezultat “neaşteptat “ :

```
x*10=1
Fals!  x*10 !=  1
```

Acest rezultat se datorează faptului că numarul 0.1 atribuit lui x are reprezentarea binară 0.00011001100110011...=0.0(0011), deci nu poate fi memorat

exact. Cu toate acestea, prin mecanismul de rotunjire utilizat de funcţia *cout*, x*10 este afişat ca fiind egal cu 1.

2) Ce rezultat furnizează programiul urmator?

```
#include<iostream.h>
void interschimba(int x,int y)
 { int aux;
   aux=x;
   x=y;
   y=aux;
 }
void main()
 { int a=10,b=20;
   interschimba(a,b);
   cout<<"a="<<a<<", b="<<b<<endl;
 }
```

Comentariu. Programul afisează:
a=10, b=20
deci valoarea variabilei *a* nu este interschimbată cu a lui *b*.

Acest lucru se datorează faptului că parametrii *x* şi *y* sunt parametri de intrare, ceea ce înseamnă că se transferă funcţiei *interschimba* valoarile parametrilor efectivi *a* şi *b*. Modificarea valorilor variabilelor *x* şi *y* de către funcţia *interschimba* nu are efect asupra conţinutului variabilelor *a* şi *b* din programul apelant.

3) Să se determine numerele de forma abxcy divizibile cu 19; x şi y se introduc de la tastatură.

Soluţie.

```
#include<iostream.h>
void main()
{ long abxcy;
  int a,b,x,c,y;
  cout<<"x,y="; cin>>x>>y; // tastati doua cifre separate prin spatiu
  for(a=1;a<=9;a++)
   for(b=0;b<=9;b++)
    for(c=0;c<=9;c++)
      { abxcy=10000*a+1000*b+100*x+10*c+y;
if(abxcy%19==0) cout<<abxcy<<"  ";
      }
  cout<<endl;
}
```

Comentariu. Programul afişează “ciudat” şi câteva numere negative. Deoarece 10000 este o constantă de tip int şi a este tot de tip int rezultă că 10000*a este tot de tip int, dar dacă a >=4 atunci 10000*4 >32567 şi astfel depăşim domeniul numerelor de tip int. Putem corecta programul, de exemplu, forţand tipul constantei 10000 la tipul long sau la tipul unsigned long folosind una din variantele:

```
abxcy=10000l*a+1000*b+100*x+10*c+y;
abxcy=long(10000)*a+1000*b+100*x+10*c+y;
abxcy=10000ul*a+1000*b+100*x+10*c+y;
abxcy=(unsigned long)10000*a+1000*b+100*x+10*c+y;
```

4) *Programul următor calculează C_n^k, n şi k se introduc de la tastatură.*

$$C_n^k = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \ldots \cdot \frac{n-k+1}{k} = \prod_{i=1}^{k} \frac{n-i+1}{i}$$

```
#include<iostream.h>
void main()
 { long C,n,k,i;

    // Citirea lui n si k:
   cout<<"n="; cin>>n;
   cout<<"k="; cin>>k;
    // Calculul combinarilor:
   C=1;  i=1;
   while(i<=k)
     { C*=(n-i+1)/i;
       i++;
     }
      // Afisarea rezultatului:
    cout<<"C("<<n<<","<<k<<")="<<C<<endl;
  }
```

Comentariu. Pentru n=10, k=3 programul afişează

```
C(10,3)=80
```

Dar rezultatul corect este 120.

Expresia `C*=( n-i+1)/i,` cu `C, n` şi `i` de tip întreg, nu produce acelaşi rezultat cu `C=C*( n-i+1)/i,` de exemplu pentru C=10, n=10, i=2 a doua expresie furnizează rezultatul C=10*9/2=45, iar prima C*=9/2 ⇒C*=4 ⇒ C=40. (`C*=( n-i+1)/i` este echivalentă cu `C=C*(( n-i+1)/i)`).

Programul se corectează înlocuind secvenţa:

```
   while(i<=k)
     { C*=(n-i+1)/i;
       i++;
     }
```

cu

```
   while(i<=k)
     { C=C*(n-i+1)/i;
       i++;
     }
```

I.9. Pointeri

Pointerii se utilizează pentru a face referire la date prin adresele lor.

Într-o variabilă de tip pointer putem păstra adresa unei date în loc de a memora data însăşi sau putem păstra adresa unei funcţii. Schimbând adresa memorată în pointer, putem manipula informaţii din diverse locaţii de memorie.

Ca orice tip de variabilă, înainte de a fi utilizată, variabila de tip pointer trebuie declarată.

Pointerii oferă posibilitatea de a aloca dinamic memoria, ceea ce înseamnă că pe parcursul execuţiei unui program se pot aloca şi dealoca zone de memorie asociate lor.

O variabilă de tip pointer se declară utilizând formatul:

```
<tip>* <nume>;
```

ceea ce înseamnă că *<nume>* este un pointer către o zonă de memorie ce conţine o dată de tipul *<tip>*.

Caracterul "*" poate fi alăturat de *<tip>* sau de *<nume>* sau poate fi separat prin caractere spaţiu şi de *<tip>* şi de *<nume>*. El indică compilatorului că a fost declarată o variabilă pointer şi nu una obişnuită.

Construcţia *<tip>** se spune ca reprezintă tipul *pointer*.

Exemplul 1:

```
int *p;
```

Aici se stabileşte faptul că `p` va conţine adrese de zone de memorie alocate datelor de tip `int`.

Exemplul 2:

```
float *p, t, *q;
```

Aici `p` şi `q` sunt pointeri către date de tip float, iar t este o variabilă de tip float.

Utilizarea pointerilor se face cu doi operatori unari:

`&` - *operatorul adresa (de referentiere)* - pentru aflarea adresei din memorie a unei variabile;

`*` - *operatorul de indirectare (de deferentiere)* - care furnizează valoarea din zona de memorie spre care pointează pointerul operand.

Dacă `x` este o variabilă atunci operatorul unar `&` aplicat lui `x`, `&x`, ne furnizează adresa lui `x`. Dacă dorim ca pointerul `p` să indice pe `x`, putem utiliza atribuirea:

```
p=&x;
```

Dacă p este o variabilă de tip pointer atunci operatorul unar * aplicat lui `p`, `*p`, ne furnizează variabila a cărei adresă este memorată în `p`.

Exemplu:

```
int a,*adr;
adr=&a;   // acum a şi *adr reprezintă aceeaşi dată.
a=100;    // este echivalentă cu: *adr=100;
*adr=200; // este echivalentă cu: a=200;
```

Există cazuri în care dorim ca un pointer să fie utilizat cu mai multe tipuri de date. În acest caz, la declararea lui nu dorim să precizăm un tip anume. Aceasta se realizează astfel:

```
void *<nume>;
```

Utilizarea tipului `void*` implică conversii explicite de tip.

Exemplu:

```
void *p;
int x;

p=&x; // Atribuire neacceptată deoarece tipul pointerului p este nedeterminat

(int*)p=&x; //Atribuire corectă: tipul void* este convertit spre int*

*p=10;  // Atribuire neacceptată deoarece tipul pointerului p este nedeterminat

*(int*)p=10; // Atribuire corectă: tipul void* este convertit spre int*
```

Deoarece pointerii reprezintă adrese, ei se folosesc la transferul prin referinţă al parametrilor.

Exemplu:

```
#include<iostream.h>
void interschimba(int *x,int *y)
  { int aux=*x;
    *x=*y;
    *y=aux;
  }

void main()
  { int a,b;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    interschimba (&a,&b);
    cout<<"a="<<a<<"  b="<<b<<endl;
  }
```

În funcţia *interschimba*, parametrii formali `x, y` sunt pointeri la `int`, astfel încât la apel, parametrii actuali trebuie să fie adrese ale unor variabile de tip `int`, şi nu valori întregi. Orice modificare se va face la adresele transmise ca parametri actuali.

O variabilă pointer poate fi iniţializată cu adresa unei variabile, astfel:

```
int x=10;
int *p=&x; // pointerul p se iniţializează cu adresa variabilei x
```

Un pointer la caractere poate fi iniţializat ca în exemplul următor:

```
char *q="abc";      // q memorează adresa  de unde începe şirul "abc"
```

Operaţii cu pointeri

Asupra pointerilor se pot face următoarele operaţii: atribuire, comparare, adunare, scădere, incrementare, decrementare.

Adunarea şi scăderea unui întreg dintr-un pointer.

Dacă `p` este un pointer având declaraţia:

```
<tip> *p;
```

atunci `p+n` furnizează valoarea lui `p` mărită cu `n*sizeof(<tip>)`, iar `p-n` valoarea lui `p` micşorată cu `n*sizeof(<tip>)`.

Asupra pointerilor se pot face operaţii de incrementare şi decrementare:

`p++` şi `++p` măresc adresa conţinută de `p` cu `sizeof(<tip>)`

`p--, --p` micşorează valoarea pointerului `p` cu `sizeof(<tip>)`.

Exemple:

```
char *c;
int *k;
float *f1,*f2;
double *d;
c++;       //c= (adresa memorată în c) + 1
k+=5;      // k=(adresa memorată în k) + 5*2
f2=f1-5;   // f2=(adresa memorată în f1) - 5*4
d-=3;      //d=(adresa memorată în d) - 3*8
d--;       //d=(adresa memorată în d) - 1*8
```

Operaţiile de incrementare şi decrementare se pot aplica pointerului însuşi sau obiectului pe care-l punctează (memorează).

Instrucţiunea *a++ obţine mai întâi valoarea pe care o punctează a şi apoi a este incrementat pentru a puncta elementul următor.

Instrucţiunea (*a)++ incrementează obiectul pe care-l punctează a.

Legătura dintre pointeri şi tablouri

Numele unui tablou este un pointer şi el are ca valoare adresa primului său element.

Fie

```
int t[5];
int *p;
p=t;  // Atribuire corectă!
```

`p` va pointa spre primul element al tabloului t. Între `p` şi `t` există o diferenţă şi anume: valoarea lui `p` poate fi modificată dar a lui `t` nu poate fi modificată (`t` este un pointer constant), deci este interzisă o atribuire de forma `t=p`;

Un nume de tablou poate fi utilizat ca şi cum ar fi un pointer şi, reciproc, un pointer poate fi indexat ca şi cum ar fi un tablou.

Dacă `x` este un tablou:

```
<tip> x[<dim>];
```

atunci expresia `x+n` este corectă şi reprezintă un pointer către al `n`-lea element al tabloului (x+n==&x[n]), deci `x[n]` este echivalentă cu `*(x+n)`.

Dacă p este un pointer:

```
<tip> *p;
```

atunci *(p+i) este echivalentă cu p[i].

O diferenţă între pointer şi tablou constă în alocarea de memorie. În cazul tabloului, se rezervă automat spaţiul necesar. În cazul pointerilor, spaţiul trebuie creat explicit de utilizator sau trebuie atribuită pointerului o adresă a unui spaţiu deja alocat.

Fie declaraţia

```
int t[5];
```

Elementele tabloului t sunt memorate în celule succesive de memorie şi sunt numerotate începând cu zero.

În exemplul nostru vom avea:

```
t[0], t[1], t[2], t[3], t[4].
```

Deoarece t[0] este o variabilă simplă, adresa sa este &t[0], deci vom avea:

```
t==&t[0].
```

Elementele unui tablou multidimensional sunt memorate în ordinea crescătoare a liniilor.

De exemplu, elementele tabloului descris prin:

```
int a[2][3];
```

vor fi memorate în ordinea

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2].
```

a reprezintă adresa primului element din tablou. Tabloul a, având două linii, este considerat ca un tablou cu două elemente (tablouri unidimensionale): elementele a[0] şi a[1]. a este adresa elementului a[0], adică avem a==&a[0]. Cum a[0] este şi el un tablou, înseamnă ca a[0] este adresa primului său element, deci a[0]==&a[0][0]. Dar, cele două tablouri, tabloul bidimensional a[][], şi tabloul liniar a[0] încep de la aceeaşi adresă, astfel că:

```
a==&a[0]==a[0]==&a[0][0]
```

Dacă privim tabloul a ca o variabilă structurată, atunci adresa acestei variabile se poate afla cu &a, deci avem lanţul de egalitaţi:

```
&a==a==a[0]==&a[0]==&a[0][0].
```

Compararea a doi pointeri

Dacă doi pointeri pointează spre elementele aceluiaşi tablou, pot fi comparaţi folosind operatorii de relaţie şi egalitate.

Astfel

```
<tip> t[dim];
<tip> *p,*q;
```

Dacă p pointează spre t[i] şi q spre t[j], atunci

p<q dacă i<j,

p!=q dacă i≠j.

Un pointer mai poate fi comparat cu constanta NULL (zero binar) utilizând operatorii == şi !=. Astfel stabilim dacă o variabilă pointer conţine sau nu o adresă.

Doi pointeri care pointează elementele aceluiaşi tablou *pot fi scăzuţi*, astfel dacă p pointează pe t[i] şi q pe t[i+n], atunci q-p are valoarea n.

Observaţie. Nu se admite adunarea a doi pointeri.

Tablouri de pointeri

Datele de tip pointer pot fi organizate în tablouri la fel ca şi alte tipuri de date. Pentru a descrie un tablou de pointeri se foloseşte o construcţie de forma:

```
<tip> *<nume>[<dim>];
```

unde <*nume*> este un tablou având <*dim*> elemente de tip pointer ce memorează adrese ale unor date de tipul <*tip*>.

Exemplu:

```
#include<iostream.h>
int DenumireZi(int m)
  { static char *zi[7]={"luni","marti","miercuri",
                "joi","vineri","sambata","duminica"};
    if (m<1||m>7) return 0; else {cout<<zi[m-1]; return 1;}
  }
```

Pentru memorarea denumirilor zilelor s-ar fi putut folosi un tablou cu două dimensiuni care să păstreze pe fiecare linie câte un şir de caractere corespunzător numelui zilei.

```
char nume[7][10]={"luni","marti","miercuri","joi","vineri",
                  "sambata","duminica"};
```

Soluţia cu pointeri are avantajul că liniile tabloului pot fi de lungimi diferite, conducând la o reprezentare eficientă a datelor.

În exemplul de mai jos, funcţia zi returnează un pointer către un şir de caractere:

```
#include<iostream.h>
char* zi(int m)
 { static char *z[7]={"luni", "marti", "miercuri",
                 "joi","vineri","sâmbată","duminică"};
   if(m<1 || m>7) return 0; //pointer NULL
   return z[m-1];
 }
void main()
 { int n;
   char *pZi;
   cin>>n;
   pZi=zi(n);
   cout<<pZi<<endl;
 }
```

I.9.1. Exemple de funcţii de lucru cu şiruri de caractere. Varianta cu pointeri.

1) Determinarea numărului de caractere dintr-un şir de caractere:

```
int lungime(char *sir)
```

```
 {
   for(int k=0; *sir++; k++);
   return k;
 }
```

2) Copierea unui şir de caractere:

```
void copiaza(char *sursa, char *dest)
   { while(*dest++=*sursa++);  }
```

```
#include<iostream.h>
void main()
  { char a[25],b[25];
    cout<<"Sirul a=";  cin>>a;
    copiaza(a,b);
    cout<<"Sirul a are "<<lungime(a)<<"  caractere."<<endl;
    cout<<"Sirul b este "<<b<<endl;
  }
```

3) Funcţie pentru transformarea în majuscule a literelor unui sir de caractere

```
#include  <stdio.h> // pentru puts, gets si printf
#include <conio.h> // pentru clrscr si getche
char* majuscule(char *s)
 { char *p=s;
   while(*p){ if(*p>='a'&&*p<='z') *p+='A'-'a';
              p++; }
   return s;
 }
void main ( )
{ char sir[1000] ;
  clrscr( );
  puts(" Introduceti sirul:");  gets(sir);
  puts(majuscule(sir));
  getche( );
}
```

Observaţii:

-Incrementarea unui pointer este mai rapidă decât indexarea unui tablou.

-Biblioteca <*string.h*> conţine funcţii pentru determinarea lungimii unui şir de caractere, pentru compararea, localizarea şi copierea şirurilor, pentru transformarea caracterelor în majuscule sau minuscule(*strlen*, *strcmp*, *strchr, strstr, strcpy, strcat, strupr, strlwr etc.*).

I.10. Alocarea dinamică a memoriei

Necesitatea definirii în programe a datelor de tip dinamic, este dată de utilizarea mai bună a memoriei, lungimea unui program variind în funcţie de volumul datelor cu care se lucrează.

Limbajele C şi C++ îi permit utilizatorului să ceară în timpul rularii programului, în funcţie de necesităţi, să se aloce memorie suplimentară sau să se renunţe la ea.

Variabilele dinamice sunt acele variabile cărora, în mod explicit, li se alocă şi dealocă memorie şi a căror dimensiune se poate modifica pe parcursul execuţiei unui program în funcţie de opţiunile programatorului.

Zona de memorie în care se face alocarea dinamică a variabilelor se numeşte heap.

Alocarea dinamică se poate face pentru tipurile de date:

-fundamentale

-structurate: tablouri, liste, arbori etc.

Programele scrise în limbajul C standard, utilizează pentru alocarea dinamică a memoriei o familie de funcţii `malloc` şi `free`, destul de greoaie în folosire. Ele au fost păstrate în C++ doar pentru menţinerea compatibilităţii. În locul lor se folosesc operatorii `new` şi `delete`.

Operatorul `new` serveşte la alocarea dinamică a memoriei. El va returna un pointer la zona de memorie alocată dinamic. În cazul în care nu există memorie suficientă, alocarea nu va avea loc. Acest fapt se semnalează prin returnarea unui pointer `NULL` (zero binar). De aceea se recomandă ca, în cazul utilizării intensive a alocării dinamice, după fiecare utilizare a lui new să se testeze valoarea returnată.

Fie p un pointer către un tip de date, adică având o declaraţie de forma:

```
<tip> *p;
```

Operatorul `new` poate fi folosit utilizând următoarele formate:

```
p=new <tip>;
p=new <tip>(<expresie>);
p=new <tip>[<dim>];
```

În varianta 1. operatorul `new` alocă, dacă este posibil, spaţiul necesar tipului <*tip*>, şi returnează adresa zonei de memorie alocate.

În varianta 2. variabila dinamică creată cu `new` se iniţializează cu valoarea expresiei <*expresie*>.

Varianta 3. se foloseşte pentru alocarea a <*dim*> variabile dinamice de tipul <*tip*> (un tablou liniar cu <*dim*> elemente). Iniţializarea tablourilor nu este posibilă.

Exemple:

```
int *p, *q,*r;
p=new int;        // se alocă memorie pentru un întreg
q=new int(7);     // se alocă memorie pentru un întreg şi se
iniţializează variabila cu 7
r=new long[10];   // se alocă un tablou de 10 întregi
```

Dezalocarea zonei de memorie alocată cu `new`, se face cu ajutorul operatorului `delete`, cu sintaxa:

```
delete <variabila>;
```

Exemple:

```
delete p;
delete r;
```

Prezentăm mai jos trei variante de utilizare a unui vector alocat dinamic:

1.

```
#include <iostream.h>
 void main()
  { int *a,n;
    cout<<"n=";
    cin>>n;
    a=new int[n];   // alocare vector
    for(int i=0;i<n; i++)
      { cout<<"a"<<i<<"="; cin>>*(a+i);}
    cout<<"a=(";
    for(i=0;i<n; i++)
      cout<<*(a+i)<<",";
    cout<<"\b)"<<endl;
    delete a;
  }
```

2.

```
#include <iostream.h>
void main()
  { int *a,n;
    cout<<"n=";
    cin>>n;
    a=new int[n];   // alocare vector
    for(int i=0;i<n; i++)
      {cout<<"a"<<i<<"="; cin>>a[i];}
    cout<<"a=(";
    for(i=0;i<n; i++)
      cout<<a[i]<<",";
    cout<<"\b)"<<endl;
    delete a;
  }
```

3.

```
#include <iostream.h>
void main()
 { int *a, n;
   cout<<"n="; cin>>n;
   a=new int[n]-1;   // astfel, elementele vectorului sunt a[1], a[2], ..., a[n]
   for(int i=1;i<=n; i++) { cout<<"a"<<i<<"="; cin>>a[i]; }
   cout<<"a=(";
   for( i=1;i<=n; i++) cout<<a[i]<<",";
   cout<<"\b)"<<endl;
   a++; // revenire la adresa returnată de new
   delete a;
 }
```

I.11. Tipul referinţă

Pentru a simplifica lucrul cu pointeri, în C++ a fost introdus tipul *referinţă*. Tipul *referinţă* implementează perfect conceptul de transmitere a parametrilor prin referinţă.

O *referinţă* este un nume alternativ pentru un obiect.

Dacă avem tipul de dată `T`, prin `T&` sau `T &` (cu spaţiu după `T`) sau `T &` (cu spaţiu după `T` şi după `&`) vom înţelege o referinţă (o trimitere) la un obiect de tipul `T`.

Exemplu:

```
int x=10;
int& r=x; // r şi x referă acum  acelaşi obiect
r=20;     // echivalent cu x=20;
x=30;     // echivalent cu r=30;
```

Variabila `r` de mai sus, este o referinţă la variabila `x` de tip `int`. Acest lucru înseamnă că identificatorii `r` şi `x` permit accesul la aceeaşi zonă de memorie. Prin urmare, `x` şi `r` sunt sinonime. Iniţializarea unei *referinţe* (trimiteri) în declaraţia sa este obligatorie (dacă nu este folosită ca argument al unei funcţii), dar această iniţializare nu trebuie confundată cu atribuirea; ea defineşte pur şi simplu un alt nume (un alias) al obiectului cu care a fost iniţializată. În exemplul de mai sus `r` este un nou nume pentru `x`. O referinţă nu mai poate fi modificată după iniţializare. Ea referă întotdeauna acelaşi obiect stabilit prin iniţializare să-l desemneze. Pentru a obţine un pointer la obiectul desemnat de referinţa `r`, se poate folosi `&r`.

Referinţele sunt utile şi când sunt folosite ca argumente pentru funcţii.

Exemplu:

```
#include<iostream.h>
void interschimba(int& a,int& b)
  {int aux=a;  a=b;  b=c; }

void main()
  { int x=10,y=20;
    interschimba (x,y);
    cout<<"x="<<x<<"y="<<y;
  }
```

Semantica transmiterii argumentelor este aceea a iniţializării, la apel argumentele `a` şi `b` ale funcţiei *interschimba*, devin alte nume pentru variabilele `x` şi `y` şi de aceea operaţiile se fac direct asupra variabilelor `x` şi `y`.

Dacă tipul unei funcţii este o referinţă, atunci acea funcţie va întoarce o variabilă. În astfel de situaţii variabila returnată trebuie să fie statică sau alocată cu operatorul `new` ca mai jos:

Exemple:

```
int& f()
  { static int x;
    ...
    return x;
```

```
  }
```

Funcţia `f()` întoarce variabila de tipul `int` cunoscută în interiorul funcţiei prin identificatorul `x`. Au sens, `f()++` (incrementează variabila returnată) şi `&f()` (reprezentând adresa variabilei returnate).

```
int& g()
 { int& x= *new int;   //Am dat un nume variabilei anonime *new int
   ...
   return x;
 }
```

Funcţia `g()` întoarce variabila de tipul int alocată dinamic.

I.11.1. Probleme rezolvate

1) Intersecţia, reuniunea şi diferenţa a două mulţimi

```
#include<iostream.h>
#include<conio.h>
void citire(int a[],int& n)
  { cout<<"numar de elemente:";cin>>n;
    int i;
    cout<<"tastati "<<n<<" elemente:";
    for(i=0;i<n;i++)cin>>a[i];
  }
void afisare(int a[],int n)
  { int i;
    cout<<"{";
    for(i=0;i<n;i++)cout<<a[i]<<" ";
    cout<<"}";
  }
void intersectie(int a[],int m,int b[],int n, int c[],int& p)
  { int i,j;
    p=0;
    for(i=0;i<m;i++)
     for(j=0;j<n;j++)
       if(a[i]==b[j]) { c[p++]=a[i];break;}
  }
void reuniune(int a[],int m,int b[],int n,int c[],int& p)
  { int i,j;
    for(i=0;i<m;i++)c[i]=a[i];
    p=m;
    int semafor; // semafor=1 daca  b[j] apartine lui a  si semafor =0
daca nu apartine
    for(j=0;j<n;j++)
      { semafor =0; // b[j] nu apartine lui a
       for(i=0;i<m;i++)
         if(b[j]==a[i]) { semafor=1; break;}
       if(semafor ==0)c[p++]=b[j];
      }
```

```
  }

void diferenta(int a[],int m,int b[],int n,int c[],int& p)
  { int i,j;
    p=0;
    int semafor;
    for(i=0;i<m;i++)
      { semafor =0;
        for(j=0;j<n;j++)
         if(a[i]==b[j]) { semafor =1;break;}
        if(semafor ==0)c[p++]=a[i];
      }
  }

void main()
  { int u[100],v[100],w[100];
    int n1,n2,n3;
    cout<<"Multimea U:"<<endl;
    citire(u,n1);
    cout<<"Multimea V:"<<endl;
    citire(v,n2);
    int op;
    do
      { clrscr();
        cout<<"U=";afisare(u,n1);cout<<endl;
        cout<<"V=";afisare(v,n2);cout<<endl;
        cout<<"1 - intersectie"<<endl;
        cout<<"2 - reuniune"<<endl;
        cout<<"3 - diferenta"<<endl;
        cout<<"4 - STOP"<<endl;
        cout<<"Tastati optiunea:";  cin>>op;

        switch(op)
         { case 1:intersectie(u,n1,v,n2,w,n3);
                  cout<<"U intersectat cu V=";
                  afisare(w,n3); cout<<endl;
                  break;
           case 2:reuniune(u,n1,v,n2,w,n3);
                  cout<<"U reunit cu V=";
                  afisare(w,n3);cout<<endl;
                  break;
           case 3:diferenta(u,n1,v,n2,w,n3);
                  cout<<"U-V="; afisare(w,n3); cout<<endl;
                  break;
         }
        getch();
      }
    while(op!=4);
  }
```

2) Algoritmul extins al lui Euclid.

Algoritmul lui Euclid calculează cel mai mare divizor comun a două numere întregi a şi b. Cel mai mare divizor comun d este cel mai mare număr natural care divide pe a şi pe b. Algoritmul lui Euclid are la bază următoarea proprietate:

Dacă $d \vee a$ şi $d \vee b$ atunci $d \vee (m \cdot a + n \cdot b) \forall\, m, n \in Z$.

Algoritmul *extins* al lui Euclid rezolvă următoarea problemă:

Fiind daţi doi întregi a şi b care nu sunt simultan 0, să se determine cel mai mare divizor comun al lor *d* şi doi întregi *m* şi *n* astfel încât $m \cdot a + n \cdot b = d$ (Identitatea lui Bézout).

Pentru obţinerea identităţii $m \cdot a + n \cdot b = d$, se pleacă de la relaţiile:

$1 \cdot a + 0 \cdot b = a\,(1)$

$0 \cdot a + 1 \cdot b = b\,(2)$, dacă $a \geq 0$ şi $b \geq 0$.

Dacă $a < 0$, relaţia (1) se înlocuieşte cu $-1 \cdot a + 0 \cdot b = -a\,(1')$.

Dacă $b < 0$, relaţia (2) se înlocuieşte cu $0 \cdot a + (-1) \cdot b = -b\,(2')$.

Cum cel mai mare divizor comun al două numere nu se modifică dacă din numărul cel mai mare scădem numărul mai mic, deducem că prin repetarea unui proces de reducere prin scădere aplicat între relaţiile (1) şi (2) ajungem în situaţia în care una din relaţii are membrul drept egal cu zero. Cealaltă relaţie ne furnizează cel mai mare divizor comun şi coeficienţii m şi n ai identităţii lui Bezout.

```
long euclid(long a,long b,long &m, long &n)
  { long m1,n1;
    if(a>=0) { m=1; n=0;} else {m=-1; n=0; a=-a;}
    if(b>=0) {m1=0; n1=1;}  else {m1=0;n1=-1;b=-b;}
    if(a==0) {m=m1; n=n1; return b;}
    while(b!=0)
      { if(a>b) { m-=m1; n-=n1; a-=b;  }
        else    { m1-=m; n1-=n;  b-=a; }
      }
     return a;
  }
```

Putem optimiza algoritmul astfel: dacă a=bq+r , 0≤r<b scădem din relaţia (1) relaţia (2) multiplicată cu q.

```
long euclid1(long a,long b,long &m, long &n)
  { long m1=1,n1=0,u,v;
    m=0; n=1;
    long q=a/b, r=a%b;
    while(r)
      { u=m1,v=n1; m1=m,n1=n;
        m=u-q*m; n=v-q*n;
        a=b; b=r; q=a/b; r=a%b;
      }
    return b;
  }
```

In limbajul C, $a\%b=r\in(-b,b)$, de exemplu pentru `a=-65, b=10` avem `a/b=-6, a%b=-5`. Algoritmul *euclid1* generează rezultate corecte şi pentru a<0 sau/şi b<0.

3) Să se determine o progresie aritmetică formată din k termeni, toţi numere prime[1].

Soluţie.

Fie $2, 3, 5, \ldots, p_{n-1}$ şirul format din primele n-1 numere prime în ordinea lor crescătoare. Dacă în acest şir nu există o progresie aritmetică formată din k termeni, generăm următorul număr prim p_n şi testăm dacă acesta este ultimul termen al progresiei căutate. Raţia progresiei aritmetice, dacă există, este egală cu $r=p_n-p_{n-1}$ sau $r=p_n-p_{n-2}$, etc. Odată fixată raţia r, testăm dacă numerele $a_1=p_n-(k-1)\cdot r, a_2=a_1+r, \ldots, a_{k-1}=a_{k-2}+r, a_k=p_n,$ sunt toate numere prime. O condiţie iniţială este ca $a_1\geq 3$.

Putem optimiza procesul de căutare dacă ţinem cont de proprietatea lui Cantor "Dacă $a_1, a_2, \ldots, a_k$este o progresie aritmetică formată din numere prime, atunci raţia ei se divide la produsul numerelor prime mai mici decât k".

Justificarea acestei proprietăţi se bazează pe următoarele observaţii:

Fie q un număr prim, 1<q<k,

- cel mult a_1=q,
- termenii progresiei $a_2=a_1+r, a_3=a_1+r, \ldots, a_{q+1}=a_1+q\cdot r$ prin împărţire la q vor furniza q valori în mulţimea {1, 2, …, q-1} de unde rezultă că există $1\leq i<j\leq q$ cu $a_i\equiv a_j(\mathrm{mod}\, q)$, adică $q\vee(j-i)\cdot r$;

cum j-i<q si q prim $\Rightarrow q\vee r$.

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
int prim(long m) // returneaza 1 daca m este prim si 0 in caz contrar
 { if(m==1) return 0; // 1 nu este prim
   if(m==2||m==3) return 1; // 2 si 3 sunt nr. prime
   if(m%2==0) return 0;
   long r=sqrt(m);
   for( long d=3; d<=r; d+=2)
     if(m%d==0) return 0;
   return 1;
 }
long nrPrimUrmator(long m)// determina cel mai mic nr prim mai
mare decat m
```

[1] In 1963 matematicianul român Solomon Marcurs a demonstrat că nu există o progresie aritmetică cu numar infinit de termeni, toți numere prime (Automates finis, progressions arithmetiques et grammaires a un nombre fini d'etats. Comptes rendus de l'Academie des Sciences, Paris, vol. 256, 1963, nr. 17, p. 3571-3574.).

Pâna acum, prin algoritmi euristici sofisticați s-au determinat progresii aritmetice formate din maxim 26 termeni numere prime.

```
 { while(!prim(++m));
   return m;
 }
void PAP(int k,long &a1,long &ratia)
   // a1 primul termen al progresiei
{ long pn;// ultimul nr prim testat
  int q=2;
  int p=2;
  p=nrPrimUrmator(p);
  while(p<k) {q*=p; p=nrPrimUrmator(p);}
     //q=produsul nr. prime mai mici decat k
  pn=(k-1)*q; //pn > (k-1)*q
  int k1;
  while(1)
  { pn=nrPrimUrmator(pn); // urmatorul nr prim

    int f=0; // f=factor de multiplicare
    while(1)
    { f++;
      ratia=f*q; // valoarea posibila a ratiei este multiplu de q
      a1=pn-(k-1)*ratia;
          // valoarea primului termen al progresiei, care urmeaza sa fie
validata
      if(a1<3) break;
      k1=1;
      for(long p=a1;p<pn ; p+=ratia)
       { if(!prim(p)) break;
         k1++;
       }
      if(k1==k) return; // progresia contine k termeni nr prime
     }
   }
 }

void main()
 { long a1, ratia;
   clrscr();
   int k;
   for(k=3;k<=12;k++)
    { PAP(k,a1,ratia);
      cout<<"k="<<k<<" solutie: a1="<<a1
          <<" ratia="<<ratia<<endl;
      cout<<a1;
      for(int i=1;i<k;i++)
   cout<<", "<<a1+i*ratia;
      cout<<endl;
    }
 }
```

Programul furnizează următoarele rezultate:
k=3, a1=3, ratia =2;
k=4, a1=5, ratia =6;

k=5, a1=5, ratia =6;
k=6, a1=7, ratia =30;
k=7, a1=7, ratia =150;
k=8, a1=199, ratia =210;
k=9, a1=199, ratia =210;
k=10, a1=199 ratia =210;
k=11, a1=110437 ratia =13860;
k=12, a1=110437 ratia =13860;

4) Programul următor determină numărul de progresii aritmetice din intervalul 2… n, formate din k termeni numere prime consecutive şi afişează progresiile cu cel puţin m termeni.

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
int nr[10];
int prim(long m)
 { if(m==1) return 0;
   if(m==2||m==3) return 1;
   if(m%2==0) return 0;
   long r=sqrt(m);
   for( long d=3; d<=r; d+=2)
     if(m%d==0) return 0;
   return 1;
 }
long nrPrimUrmator(long m)// determina cel mai mic nr prim mai mare decat m
 { while(!prim(++m));
   return m;
 }
void afisarePAPC(long a1,long ratia,int k)
 { cout<<a1;
   for(int i=1;i<k;i++)
     cout<<", "<<a1+i*ratia;
   cout<<endl;
 }
void PAPC(long n,int m)/* Contorizeaza in vectorul nr[], numarul de progresii aritmetice din intervalul 2...n, formate din k termeni numere prime consecutive si afiseaza progresiile cu cel putin m termeni */
{
  long a1=3, a2=5,r=a2-a1,an;
  //a1=primul termen al  posibilei progresii  de ratie r

  int k=2;
  while(a2<n)
   {for(an=a2+r; prim(an); an+=r)k++; /* determinarea progresiei a1, a1+r,..., a1+(k-1)r */
    if(k>=3)nr[k-3]++;
```

```
    if(k>=m)afisarePAPC(a1,r,k);

    a1+=(k-1)*r; //primul termen al  urmatoarei, posibile, progresii
    a2=nrPrimUrmator(a1); // urmatorul nr prim.
    r=a2-a1;
    k=2;
   }
}

void main()
 { clrscr();
   long n=1000000;
   PAPC(n,5);
   cout<<"in intervalul [2, "<<n<<"] exista:"<<endl;

   for(int i=0;i<10;i++)
    if(nr[i]>0)
      cout<<nr[i]<<" progresii formate din "
       <<i+3<<" termeni nr. prime consecutive"<<endl;
 }
```

5) Următorul exemplu ne arată cum putem reprezenta eficient o matrice simetrică.

O matrice simetrică de ordinul `n` este bine determinată dacă pentru fiecare linie i, i=0,1,...,n–1 se cunosc elementele: a[i][0], a[i][1], ..., a[i][i].

```
#include<iostream.h>

void alocare(int **&a,int n) // a este parametru de ieşire
 { int i,j;
   a=new int*[n];     // a este un pointer la n pointeri către întregi
   for(i=0; i<n; i++)
     a[i]=new int[i+1]; // a[i] este un pointer la i+1 întregi
 }

void dezalocare (int **a,int n)
 { int i;
   for(i=0; i<n; i++)
     delete a[i];   // vectorul pointat de a[i] este şters
  delete a;     // şterge vectorul de pointeri, indicat de a
 }
void cit(int **a,int n)
  { int i,j;
   for(i=0;i<n;i++)
     for(j=0;j<=i;j++)
       cin>>a[i][j];
   }

void scr(int **a,int n)
  { int i,j;
```

```
    for(i=0;i<n;i++)
      { for(j=0;j<=i;j++)cout<<a[i][j];
        cout<<endl;
      }
  }

 void main()
  { int **a,n=4;
    alocare(a,n);
    cit(a,n);
    scr(a,n);
    dezalocare(a,n);
  }
```

În varianta următoare, funcţia *alocare* va returna adresa la care este alocată matricea triunghiulară:

```
int** alocare(int n)
 { int **a;
   int i,j;
   a=new int*[n]; // a este un pointer la n pointeri către întregi
   for(i=0; i<n; i++)
     a[i]=new int[i+1]; // a[i] este un pointer la i+1 întregi
   return a;
 }
void main()
 { int n=4;
   int **a=alocare(n);
   cit(a,n);
   scr(a,n);
   dezalocare(a,n);
 }
```

I.12. Constante simbolice

Specificatorul `const` folosit într-o declaraţie, defineşte o constantă simbolică. O constantă simbolică în C++ este un nume căruia i se asociază o valoare care nu poate fi schimbată. Există trei tipuri de constante simbolice:

1) Oricărei valori de orice tip i se poate da un nume şi poate fi folosită ca o constantă, dacă definiţia sa este prefixată de cuvântul cheie `const`;

2) Un set de constante întregi se poate defini ca o enumerare;

3) Orice nume de vector sau de funcţie este o constantă.

Comentarii:

Pentru că nu poate fi modificată o constantă trebuie să fie iniţializată.

Exemple:

```
const int h=176;
const int v[]={1,2,3,4};    //vector constant
```

Dacă tipul unei constante simbolice nu este precizat, se consideră că acea constantă are tipul `int`. Când se foloseşte un pointer, sunt implicate două obiecte: pointerul însuşi şi obiectul spre care el indică. Dacă se prefixează declaraţia unui pointer cu const, obiectul şi nu pointerul este constant.

De exemplu:

```
const char *p ="abcd"; // p este un pointer la o constantă
p[3]='a';          // eroare !
p="efgh";          // p indică spre alt obiect
```

Pentru a declara că însuşi pointerul şi nu obiectul spre care el indică este o constantă, se foloseşte construcţia `*const`.

De exemplu:

```
char * const p ="abcd";   // p este un pointer constant.
p[1]='x';                 // ok! şirul nu este constant
p="efgh";                 // eroare
```

Pentru a face ambele obiecte constante, le declarăm pe amândouă cu `const`, astfel:

```
const char * const p="abcd"; // p este pointer constant la o
constantă
p[1]='x';   // eroare !
p="efgh";   // eroare !
```

Adresa unei constante nu poate fi atribuită unui pointer, pentru a nu se permite ca valoarea obiectului să fie modificată.

I.12.1. Parametri constanţi pentru securitatea codului

```
float raport(int num,int den)
 { if(den=0) return num;
   return (float)num/den ;
 }
```

Funcţia de mai sus conţine o eroare de programare pe care compilatorul nu o detectează. Într-adevăr, den = 0 este o atribuire şi nu un test de egalitate. Parametrul den este adus sistematic la valoarea 0. Cum putem preveni astfel de erori? Putem utiliza modificatorul *const* pentru parametrii de intrare care nu trebuie să evolueze în funcţie.

```
float raport(const int num,const int den)
 { if(den=0) return num; // error: Cannon modify a const object
   else return (float)num/den ;
 }
```

În acest caz compilatorul indică faptul că a detectat o eroare: un obiect constant (care nu poate fi modificat) este utilizat în partea stânga a unei atribuiri.

I.13. Tipul enumerare

Acest tip permite folosirea unor nume sugestive pentru valori numerice întregi.

O enumerare reprezintă o listă de constante întregi, pusă în corespondenţă cu o listă de identificatori.

Constantele întregi sunt adesea definite mai convenabil cu `enum`.

Exemplu:

```
enum{ IAN, FEB, MAR, APR, MAI};
```

defineşte patru constante întregi numite enumeratori şi le atribuie valori.

Deoarece valorile de enumerare sunt atribuite implicit începând de la zero, declaraţia de mai sus este echivalentă cu:

```
const int IAN=0, FEB=1, MAR=2, APR=3, MAI=4;
```

Dacă o enumerare poartă un nume, ca în exemplul de mai jos, acel nume devine sinonim cu `int`, nu este un nou tip.

Enumeratorii pot primi valori explicite care nu trebuie să fie distincte crescătoare sau pozitive.

Exemple:

```
enum taste {stanga=4,dreapta=6,sus=8,jos=2};
taste t;
...
t=dreapta;
...
```

```
enum culori {galben,albastru=5,rosu,verde=-5,alb} CULOARE;
```

În acest caz avem echivalenţa cu:

```
const galben=0, albastru=5, rosu=6, verde=-5, alb=-4;
```

În acest exemplu CULOARE este o variabilă de tip asumat `int`, deci putem scrie:

```
CULOARE=rosu;
```

sau

```
CULOARE=6;
```

sau

```
CULOARE=100;
```

I.14. Prototipul unei funcţii

În principiu, o funcţie poate fi apelată dacă este definită în fişierul sursă înainte de a fi apelată. Acest lucru nu este întotdeauna posibil şi în astfel de cazuri apelul funcţiei trebuie să fie precedat de prototipul ei.

Prototipul unei funcţii are ca scop să informeze compilatorul despre:

- tipul valorii returnate de funcţie
- tipurile parametrilor.

În felul acesta, la apelul unei funcţii, compilatorul poate face teste cu privire la tipul expresiilor parametrilor efectivi precum şi unele conversii necesare legate de valoarea returnată de funcţie.

Prototipul unei funcţii poate fi evidenţiat utilizând formatele:

```
<tip> <nume funcţie>(<lista parametrilor formali>);

<tip> <nume funcţie>(<lista tipurilor parametrilor formali>);
```

Exemple:

```
1.  int f1(float x,double a);
      sau
    int f1(float,double);

2.  void f2(int a[],int b[]);
      sau
    void f2(int [],int []);

3.  long f3(int x[4][4]);
      sau
    long f3(int [][]);
```

```
4.
#include <iostream.h>
void f(int [][]);

void main()
  { int a[][2]={0,1,0,2};
    f(a);
  }

void f(int x[][2])
  { cout<<x[1][1]; }
```

Exemplul 4 ne arată că, în prototip, pentru parametrii formali de tip tablou pot lipsi toate dimensiunile, dar în antetul funcţiei poate lipsi numai prima dimensiune a tabloului.

I.15. Asignări de nume pentru tipuri de date

Programatorul poate să asocieze oricărui tip de date (predefinit sau utilizator), un nume utilizând construcţia `typedef` cu sintaxa:

```
typedef <tip> <nume>;
```

Exemple:

```
1.      typedef unsigned long   natural;
```

Declaraţia:

```
natural a;
```

este echivalentă cu:

```
unsigned long a ;
```

```
2.    typedef char* ptrchr;
```

Declaraţia:

```
 ptrchr s="abc",t="zxc";
```

este echivalentă cu:

```
 char *s="abc",*t="zxc";
```

3. Tipul tablou unidimensional cu 10 elemente poate fi declarat astfel:

```
typedef  int vector[10];
```

Declaraţia:

```
vector y={1,2,3,4};
```

este echivalentă cu:

```
int y[10]={1,2,3,4};
```

4. Tipul tablou bidimensional poate fi declarat ca în exemplul următor:

```
typedef  int matr[10][10];
```

Declaraţia:

```
matr x={{1,1},{1,2}};
```

este echivalentă cu:

```
int x[10][10]={{1,1},{1,2}};
```

I.16. Pointeri care memorează adrese de funcţii

În declaraţia

```
int f();
```

identificatorul `f` este un pointer care indică adresa funcţiei `f`. El este un pointer constant, deci nu poate fi modificat prin instrucţiuni de program. Putem declara variabile pointer care să memoreze adrese de funcţii în felul urmator:

```
<tip> (*<identificator pointer>)(<tipurile parametrilor>);
```

Exemplu:

```
int (*adrF)();
```

Conform acestei declaraţii, variabila `adrF` poate să memoreze o adresă a unei funcţii care calculează un rezultat de tip întreg şi are lista parametrilor formali vidă. Să observăm că parantezele care delimitează `*adrF` sunt absolut necesare în declaraţia anterioară. În lipsa lor, declaraţia `int *adrF();` descrie o funcţie care calculează un pointer la o valoare de tip `int`.

Atunci când se menţionează numai numele unei funcţii, fără lista parametrilor asociaţi şi fară paranteze se obţine adresa de început a funcţiei. Prin urmare, variabilei pointer din exemplul de mai sus i se poate atribui o valoare printr-o atribuire de forma:

```
adrF=f;
```

unde f este numele unei funcţii ce returnează un întreg şi are lista parametrilor formali vidă.

Este greşit să se scrie:

```
adrF=f();
```

sau

```
adrF=&f();
```

deoarece `f()` reprezintă valoarea returnată de funcţia f() iar &f() este adresa rezultatului furnizat de funcţie, dacă acest rezultat este o referinţă. Apelarea funcţiei a cărei adresă este memorată într-o variabilă pointer se poate face, pentru exemplul de mai sus, în una din variantele:

```
a)        adrF()
b)        (*adrF)()
```

Utilizând construcţia `typedef` putem să asignăm un nume şi tipului pointer la funcţii ca în exemplul următor:

```
typedef double (*fct)(double);
```

`fct` este identificatorul tipului de date pointer la funcţii care întorc o valoare de tip double şi au un argument de tipul double. Acum are sens, de exemplu:

```
fct f=sin;
```

Funcţia `sort` din programul următor foloseşte parametrul `ord` de tip pointer la funcţii, pentru stabilirea criteriului de ordonare crescătoare sau descrescătoare a elementelor unui vector

```
#include<iostream.h>
int cresc(int a,int b) {return a<=b;}
int descr(int a,int b) {return a>=b;}

typedef int (*criteriuOrdonare)(int,int);
enum boolean{false,true};

void sort(int x[],int n,criteriuOrdonare ord)
  { boolean sortat=false;
    int aux;
    while(!sortat)
     { sortat =true;
       for(int i=0;i<n-1;i++)
       if (!ord(x[i],x[i+1]))
{int aux=x[i]; x[i]=x[i+1]; x[i+1]=aux; sortat=false;}
       n--;
     }
  }
void main()
 { int x[5]={3,5,1,8,7};
   cout<<"Ordonare crescatoare:";
   sort(x,5,cresc);
   for(int i=0;i<5;i++) cout<<x[i]<<" ";
   cout<<endl<<"Ordonare descrescatoare:";
   sort(x,5,descr);
   for(i=0;i<5;i++) cout<<x[i]<<" ";
   cout<<endl;
 }
```

I.17. Argumente implicite pentru funcţii

C++ oferă posibilitatea declarării funcţiilor cu valori implicite pentru unele argumente. La apelarea unei astfel de funcţii, se pot omite parametrii efectivi pentru acei parametri formali care au declarate valori implicite. Un argument implicit poate fi iniţializat numai cu o expresie constantă. Se pot specifica oricâte argumente cu valori implicite. Argumentele implicite trebuie să fie ultimele în lista de argumente. Nu sunt permise alternări între argumente normale şi argumente cu valori implicite. C++ permite specificarea argumentelor implicite fie în definiţia funcţiei fie în prototip, dar nu în ambele. La apelare, lista parametrilor efectivi, corespunde primilor parametri formali.

Exemplu:

```
#include<iostream.h>
void conv(long n,int baza=2)/* afisează numărul n in baza de
numerație specificată
                                                    prin parametrul
baza( cu valoarea implicită 2) */
  { if (n<baza)
     { if(n<10) cout<<n; else cout<<"("<<n<<")"; return; }
     conv(n/baza,baza); // Apel recursiv. ( vezi I.21. Recursivitatea
în limbajul C)
     int c=n%baza;
     if(c<10) cout<<c; else cout<<"("<<c<<")";
  }

void main()
 { cout<<endl;
   int n=19;
   cout<<n<<"="; conv(n);cout<<" (baza 2)"<<endl;
   cout<<n<<"="; conv(n,8); cout<<" (baza 8)"<<endl;
 }
```

Apelul conv(n) foloseşte baza 2(implicită), iar apelul conv(n,8) foloseşte baza 8.

I.18. Funcţii supraîncărcate

C++ permite existenţa mai multor funcţii cu acelaşi nume dar cu argumente diferite ca număr sau tip.

Astfel de funcţii se numesc supraîncărcate. Compilatorul va determina funcţia apelată prin examinarea tipului argumentelor şi încercarea de a face corespondenţa cu argumentele efective. Compilatorul nu verifică tipul valorii returnate de funcţie. Deci, două funcţii supraîncărcate nu pot diferi doar prin valoarea returnată. De asemenea, nu pot avea acelaşi nume funcţiile cu argumente

ce diferă doar prin faptul că unele sunt de tip referinţă iar celelalte nu sau unele sunt statice iar celelalte nu.

```
#include<iostream.h>
long produs(int a[],int n)
  { long P=1;
    for(int i=0;i<n;i++) P*=a[i];
    return P;
  }
long produs(long a[],int n)
  { long P=1;
    for(int i=0;i<n;i++) P*=a[i];
    return P;
  }
double produs(float a[],int n)
  { long P=1;
    for(int i=0;i<n;i++) P*=a[i];
    return P;
  }
double produs(double a[],int n)
  { long P=1;
    for(int i=0;i<n;i++) P*=a[i];
    return P;
  }
void main()
{ int a[3]={1,2,3};
  float b[4]={2.,3.,4.,5.};
  cout<<produs(a,3)<<endl;
  cout<<produs(b,4)<<endl;
}
```

I.19. Funcţii inline

Dacă o funcţie este declarată `inline` compilatorul va genera codul corespunzător funcţiei în poziţia apelului, în loc de a genera secvenţa de apel. Semantica apelului rămâne neschimbată. Atributul `inline` trebuie folosit numai pentru funcţiile de dimensiuni foarte mici, pentru care regia de apel este semnificativă în raport cu timpul de execuţie al funcţiei propriu-zise. `Inline` este o cerere adresată compilatorului care poate să nu fie onorată, caz în care se generează secvenţa obişnuită de apel (de exemplu dacă în program se utilizează pointeri către funcţia respectivă).

O funcţie `inline` se declară astfel:

```
inline <tip> <nume funcţie>(<lista argumentelor>)
    { <corp functie> }
```

Observaţie. *O funcţie inline nu poate să conţină instrucţiuni repetitive.*

I.20. Structuri şi uniuni

O *structură* reprezintă o colecţie de date de tipuri diferite.

Tipul unei astfel de date se spune că este definit de utilizator şi se numeşte *tip structurat*.

În C++ o structură se poate declara utilizând sintaxa:

```
struct <identificator structură>
  {<lista de declaraţii>} <listă de variabile>;
```

<listă de variabile> poate să lipsească.

Exemplu:

```
struct  complex{float re,im;} c1,c2,c3[10];
```

Variabilele c1 şi c2 sunt structuri cu câte două câmpuri de tip *float*, iar c3 este un tablou de asemenea structuri.

În C++ putem declara variabile de tip structurat prin:

```
<identificator structura> <lista de variabile>;
```

Exemplu:

```
complex z1,z2,*p;
```

Fie declaraţia:

```
complex x,y,*p;
```

Câmpurile componente ale unui date structurate pot fi referite în două feluri:

- *direct*, prin numele structurii urmat de "." şi de numele câmpului

Exemple: x.re, x.im, y.re, y.im

- *indirect*, prin adresa structurii urmată de "–>" şi de numele câmpului

Exemple:

```
p->re,  p->im
```

Această scriere este forma simplificată a scrierii `(*p).re, (*p).im`

Operatorul `"->"` are aceeaşi prioritate ca şi `"."`. Ambii operatori sunt de prioritate maximă.

Componentele unei date structurate pot fi ele însele date structurate.

Elementele unei date de tip structurat pot fi iniţializate astfel: în declaraţie, după numele variabilei structurate se scrie "=", iar după acesta, între acolade se iniţializează componentele structurii.

Exemple:

```
complex x={1,0},y={2,1};
```

```
struct student
  { char nume[20];
    int note[10];
  };

student s1={ "Popa Dan",{10,10,10,9,10}};
```

I.20.1. Probleme rezolvate

1) În următorul program este definită structura student şi se exemplifică utilizarea ei.

```
#include<iostream.h>
#include<iomanip.h>
struct student
  { char nume[20];
    char adresa[40];
    long  telefon;
  };

void main()
 {student s[100]; // vector cu componente structurate
  int n;
  cout<<"n=";  cin>>n;
  for(int i=0;i<n; i++)
   { cin.get();       // extragerea caracterului <enter> din stream
     cout<<"student "<<i<<":"<<endl;
     cout<<" nume:";
     cin.getline(s[i].nume,30); // citeşte nume
     cout<<" adresa:";
     cin.getline(s[i].adresa,40); //citeşte adresa
     cout<<" telefon:"; cin>>s[i].telefon;
   }
  cout<<" Lista studentilor"<<endl;
  cout<<setiosflags(ios::left);   // aliniere la stânga
  for(i=0;i<n; i++)
   { cout<<setw(20)<<s[i].nume<< setw(30)<<s[i].adresa
         <<s[i].telefon<<endl;
   }
 }
```

2) Calculul ariei unui poligon cu n laturi, n≥3, când se cunosc coordonatele rectangulare ale vârfurilor poligonului (x_i, y_i)*, i=1, 2, ..., n utilizând formula:*

$$A=|\ (x_1+x_2)(y_1-y_2)+(x_2+x_3)(y_2-y_3)+\ldots+\ (x_n+x_1)(y_n-y_1)\ |/2,$$

şi determinarea coordonatelor centrului de greutate al poligonului.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

struct punct {float x,y;};

int cit( punct P[])
 { int n;
   cout<<"Nr. puncte=";  cin>>n;
```

```
   for(int i=0;i<n; i++)
    { cout<<"P"<<i+1<<"(x y)="; cin>>P[i].x>>P[i].y; }
   return n;
 }

float arie( punct P[],int n)
  { float s=(P[n-1].x+P[0].x)*(P[n-1].y-P[0].y);
    for(int i=0;i<n-1;i++)
      s+=(P[i].x+P[i+1].x)*(P[i].y-P[i+1].y);
    return fabs(s)/2;
  }

punct centruGr(punct P[],int n)
    // ne arată că o funcţie poate returna o dată structurată
  { punct G={0,0};
    for(int i=0;i<n;i++)
     { G.x+=P[i].x;  G.y+=P[i].y;  }
    G.x/=n; G.y/=n;
    return G;
  }

void main()
  { punct P[100];
    int n =cit(P);
    float S=arie(P,n);
    punct C=centruGr(P,n);
    cout<<setprecision(4);
    cout<<"Aria poligonului="<<S<<endl;
    cout<<"Centru de greutate este C("<<C.x<<","
        <<C.y<<")"<<endl;
  }
```

3) Transfer rapid de tablouri:

```
#include<iostream.h>

const dim=10;
int a[dim]={1,2,3,4,5,6,7,8,9,10},b[dim];

void main()
  {  struct vector { int v[dim];};
     *(vector*)b = *(vector*)a; // execuţia transferului
     cout<<"b=";
     for(int i=0;i<dim;i++) cout<<b[i]<<" ";
     cout<<endl;
  }
```

4) Funcţie care transferă blocuri de date:

```
void transfer(void *dest,void *sursa,int n)
  { const dimBloc=256;
```

```
    struct bloc { char a[dimBloc];};
    bloc* d=(bloc*)dest;
    bloc* s=(bloc*)sursa;
    while (n>=dimBloc)
       {  *d++=*s++; n-=dimBloc; }
    char* p1=(char*)d;
    char* p2=(char*)s;
    while (n--) *p1++=*p2++;
  }

#include<iostream.h>

void main()
   { const dim=600;
     int a[dim],b[dim];
     for(int i=0;i<dim;i++) a[i]=i;
     transfer(b,a,sizeof(a));
     for (i=0;i<dim;i++) cout<<b[i]<<" ";
     cout<<endl;
   }
```

O *uniune* este o structură de date care permite folosirea în comun a aceleiaşi zone de memorie, de două sau mai multe variabile diferite, la momente de timp diferite.

Forma generală a unei uniuni:

```
union nume_uniune
    {
     tip1 nume_câmp1;
     tip2 nume_câmp2;
        . . .
     tipn nume_câmpn;
    } lista_variabile_uniune;
```

Se observă că forma generală de declarare a unei uniuni este asemănătoare cu cea a unei structuri şi ceea ce s-a spus la structuri este valabil şi la uniuni.

Operatorul sizeof aplicat tipului de date union, adică sizeof (union nume_uniune) va furniza lungimea uniunii(lungimea celui mai mare membru al uniunii).

Deosebirea fundamentală dintre o uniune şi o structură constă în modul în care câmpurile folosesc memoria.

La structură, zonele de memorie rezervate câmpurilor sunt diferite pentru câmpuri diferite.

La uniune, toate câmpurile din uniune împart aceeaşi zonă de memorie. Aceasta înseamnă că numai valoarea unuia din câmpuri poate fi memorată la un moment dat în zona de memorie rezervată variabilei uniune.

Exemplu:

```
union h
 {
   int i;
   float t;
 }x;
```

Datele din variabila x vor fi privite ca întregi dacă selectăm x.i sau reale dacă selectăm x.t.

Câmpurile i şi t se referă la aceeaşi adresă:

în x.i se memorează sizeof(int) octeţi la această adresă;

în x.t se memorează sizeof(float) octeţi care încep la această adresă.

I.21. Recursivitatea în limbajul C

Spunem despre o funcţie C că este recursivă dacă se autoapelează. Funcţia recursivă se poate reapela fie direct, fie indirect prin apelul altor funcţii.

Funcţiile recursive se definesc prin punerea în evidenţă a două seturi de instrucţiuni şi anume:

• un set care descrie modul în care funcţionează funcţia pentru anumite valori (iniţiale) ale unora dintre argumente;

• un set care descrie procesul recursiv de calcul.

Valorile unei funcţii recursive se calculează din aproape în aproape, pe baza valorilor cunoscute ale funcţiei pentru anumite argumente iniţiale. Pentru a calcula noile valori ale unei funcţii recursive, trebuie memorate valorile deja calculate, care sunt strict necesare. Acest fapt face ca implementarea în program a calculului unor funcţii recursive să necesite un consum mai mare de memorie, rezultând timpi mai mari de execuţie.

Recursivitatea poate fi transformată în iteraţie.

In general, forma iterativă a unei funcţii este mai eficientă decât cea recursivă în ceea ce priveşte timpul de execuţie şi memoria consumată.

În alegerea căii (interativă sau recursivă) de rezolvare a unei probleme, trebuie consideraţi o serie de factori: uşurinţa programării, testării şi întreţinerii programului, eficienţa, complexitatea etc.

Dacă o problemă are o complexitate redusă este preferată varianta iterativă.

Forma recursivă este preferată acolo unde transformarea recursivităţii în iteraţie cere un efort de programare deosebit, algoritmul pierzându-şi claritatea, testarea şi întreţinerea devenind astfel foarte dificile.

La fiecare apel al funcţiei recursive, parametrii şi variabilele ei locale automatice se alocă pe stivă într-o zonă nouă, independentă. De asemenea în stivă se trece adresa de revenire în subprogramul chemător, adică adresa instrucţiunii următoare apelului. La revenire, se realizează curăţarea stivei, adică zona de pe stivă afectată la apel parametrilor şi variabilelor automatice, se eliberează.

Observaţii:

În general, recursivitatea permite o scriere mai compactă şi mai clară a programelor care conţin procese de calcul recursiv.

De obicei, recursivitatea nu conduce nici la economie de memorie şi nici la execuţia mai rapidă a programelor. În mod frecvent sunt variante nerecursive mai rapide decât variantele recursive şi conduc adesea şi la economie de memorie.

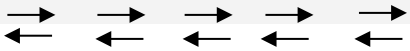
Apelurile recursive pot conduce la depăşirea stivei.

I.21.1. Probleme rezolvate:

1. Funcţia factorial: fact : N→N

$$\text{fact}(n) = \begin{cases} 1 \text{ dacă } n = 0, \\ \end{cases}$$

```
#include <stdio.h>
int fact(int n)
 { if(n==0) return 1;
   return n*fact(n-1);
 }
void main()
 { int n;
   clrscr();
   printf ("n = ") ; scanf ("%d", &n) ;
   printf ( "%d!=%d",n, fact(n));
 }
```



Apelul lui fact(4) declanşează un lanţ de apeluri ale lui fact pentru 3, 2, 1,0 după care urmează revenirea din apeluri şi evaluarea lui fact pentru 0,1,2,3,4.

fact(4)

4 fact(3)

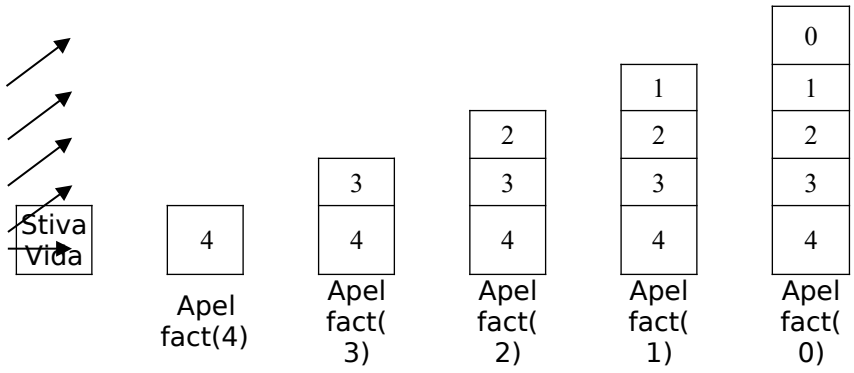
3 fact(2)

2 fact(1)

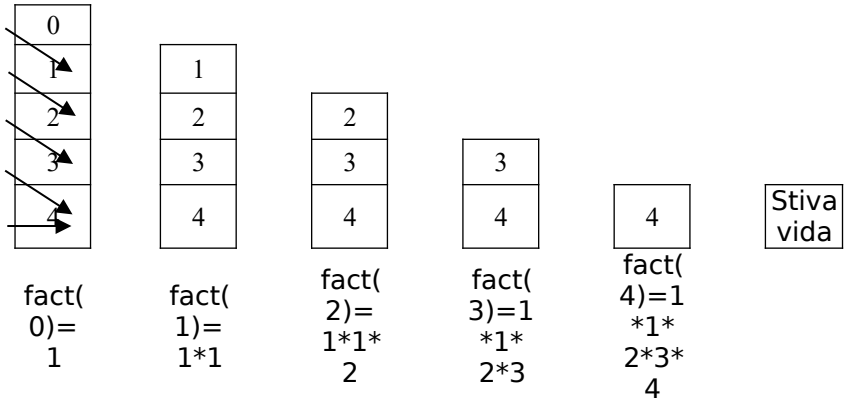
1 fact(0)

1

Starea stivei in timpul execuţiei succesive a autoapelării:



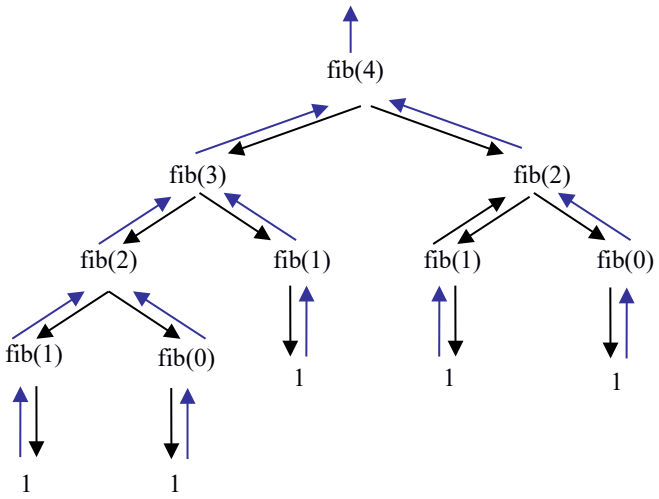
Pentru fiecare din aceste apeluri, în stivă se vor depune parametrii actuali: 4,3,2,1,0. Stările stivei după ieşirea din autoapel sunt urmatoarele:



Rezolvarea fiecărui autoapel înseamnă deplasarea pe un nivel inferior.

2. *Funcţia lui Fibonacci:* $\text{fib} : N \to N$,

$$\text{fib}(n) = \begin{cases} 1, & n = 0 \text{ sau } n = 1, \end{cases}$$




```
long fib(int n)
 { if (n==0||n==1) return 1;
   return fib(n-1) +  fib(n- 2);
 }
```

Varianta iterativa:

```
long fib(int n)
  { long f1=1,f2=1,f;
    if(n==0||n==1)return 1;
    for(k=2;k<=n;k++)
      { f=f2+f1;   f1=f2;    f2=f; }
     return f;
  }
```

3. *Funcţia lui Ackermann:* $ac : N \times N \rightarrow N$

$$\text{ac}(m, n) = \begin{cases} n+1, & \text{dacă } m = 0 \\ \text{ac}(m-1, 1), & \text{dacă } n = 0 \end{cases}$$

```
int ac(int m, int n)
 { if(m==0)return n+1;
   if(n==0) return ac(m-1,1);
   return ac(m-1,ac(m,n-1));
 }
```

4) Fie v un vector cu n elemente de tip long. Elaboraţi funcţii C recursive pentru citirea, afişarea, determinarea sumei şi determinarea minimului elementelor vectorului.

Rezolvare:

```
#include<iostream.h>

long suma(long v[],int n)
 { if(n==0) return 0;
   return v[n-1]+suma(v,n-1);
```

```
 }

long min(long v[],int n)
 { if(n==1) return v[0];
   long min2=min(v,n-1);
   return min2<=v[n-1]?min2:v[n-1];
 }

void citire(long v[],int n)
 { if(n==0)return;
   citire(v,n-1);
   cin>>v[n-1];
 }

void afisare(long v[],int n)
 { if(n==0) return;
   afisare(v,n-1);
   cout<<v[n-1]<<" ";
 }

void main()
 { long v[100];
   int n;
   cout<<"n=";
   cin>>n;

   cout<<"Tastati "<<n<<" elemente";
   citire(v,n);

   cout<<"Vectorul citit este:";
   afisare(v,n);

   cout<<endl;
   long S=suma(v,n);
   cout<<"Suma elementelor vectorului este "<<S<<endl;
   cout<<"Elementul minim este "<<min(v,n)<<endl;
 }
```

5) Calculul determinanţilor

Programul următor calculează valoarea determinantului unei matrice pătratică de ordinul n,

$$\det(A)=\begin{vmatrix} a_{00} & a_{01} & \dots & a_{0n-1} \\ a_{10} & a_{11} & \dots & a_{1n-1}a_{n-1n-1}¿ \\ \dots. & ¿\,a_{n-10} & a_{n-11} & ¿ \end{vmatrix},$$ prin utilizarea urmatoarei formule de recurenţă(dezvoltarea după prima linie):

$$\det(A)=¿ \quad ,$$

unde M_{0j} este minorul lui a_{0j}(determinantul obţinut din cel iniţial, prin suprimarea liniei 0, coloana j).

Dacă det(A)≠0 putem calcula inversa matricei A după formula $A^{-1}=\frac{1}{\det(A)}A^{\square}$,

Matricea $A^{\square}=(b_{ij}),i,j=0,1,\ldots,n-1$ se numeşte adjuncta matricei A, unde $b_{ij}=¿$ iar M_{ji}este minorul corespunzător elementului a_{ji} al matricei A.

Programul conţine şi o funcţie pentru determinarea adjunctei $A^{\square}$.

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#include <conio.h>

float ** alocare(int n)
     // alocare dinamica pentru o matrice patratica de ordin n
   { float **m=new float*[n];
     for(int i=0;i<n;i++)m[i]=new float[n];
     return m;
   }

void dezalocare(float **a,int n)
     // eliberarea memoriei alocată dinamic pentru matricea patratica de
ordin n
  { for(int i=0;i<n;i++) delete a[i];
    delete a;
  }

float ** minor(float **A,int n,int l,int k)
     // Crearea matricei minorului corespunzator elementului A[l][k]
  { float **m=alocare(n-1);
    int i,j,ii,jj;
    jj=0;
    for (j = 0 ; j < n ; j++)
      { if(j==k) continue;
        ii=0;
        for (i = 0; i < n; i++)
          { if(i==l) continue;
            m[ii][jj] = A[i][j];
            ii++;
          }
        jj++;
      }
   return m;
 }

float det(float **A, int n)
 { int i, j, p,k;
   float d;
   if (n==1)return A[0][0];
   if (n == 2) return A[0][0]*A[1][1] - A[1][0]*A[0][1];
```

```
      //altfel, dezvoltare dupa linia 0
   d = 0;
   for (p = 0 ; p < n ; p++)
     { float **m=minor(A,n,0,p);
       d += pow(-1.0, p) * A[0][p] * det(m, n-1);
       dezalocare(m,n-1);
     }
  return d;
 }

float ** adjuncta(float **A,int n)
 { float **B=alocare(n);
   float **m;
   int i,j,l,k;
   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
      { m=minor(A,n,i,j);
        B[j][i]=pow(-1,i+j)*det(m,n-1);
      }
  return B;
 }

void afisare(float **A,int n)
   // afisarea matricei patratice de ordin n
 { int i,j;
   for(i=0;i<n;i++)
     {for(j=0;j<n;j++)
       cout<<setw(8)<<A[i][j]<<" ";
      cout<<endl;
      }
 }

void main()
 { float A[4][4]={{2,1,-1,3},
                  {1,-1,2,2},
                  {4,-2,3,1},
                  {-2,2,1,-1}};
   float **m=alocare(4);
   clrscr();

     //copierea matricei A in zona alocata dinamic
   for(int i=0;i<4;i++)
    { for(int j=0;j<4;j++) m[i][j]=A[i][j];}

   cout<< "d="<<det(m,4)<<endl;
   float **b=adjuncta(m,4);

   cout<<"Adjuncta lu A:"<<endl;
   afisare(b,4);
 }
```

6) Se consideră fotografia alb-negru a unor obiecte, reprezentată sub forma unui tablou bidimensional cu n linii şi n coloane, cu elementele în mulţimea {0,1}. Elementele egale cu 1 corespund poziţiilor ce aparţin obiectelor. Să se scrie un program pentru a determina câte obiecte sunt fotografiate şi din câte elemente de "1" este compus fiecare obiect.

Rezolvare.

Cât timp sunt elemente egale cu 1, alegem un astfel de element.

Funcţia recursivă `int fig(int i, int j)`, înlocuie cu –1 elementul 1 ales şi toate elementele 1 "legate" de elementul ales (pentru a nu mai fi contorizate ulterior), şi returnează numărul poziţiilor ce aparţin obiectului curent.

Pentru simplificarea testelor matricea `a[][]` este bordată cu elemente nule.

Funcţia `void generare()`, generează o configuraţie de obiecte.

Funcţia `int figNR(int i, int j)` reprezintă varianta nerecursivă a funcţiei int fig(int i, int j). Eliminarea recursivităţii s-a făcut prin utilizarea unei stive formată cu ajutorul vectorilor `x[]` şi `y[]`. Vârful stivei este indicat de indicele k; `(x[k],y[k])` reprezintă poziţia unui element 1 ce aparţine obiectului curent.

Iniţial în stivă se introduce poziţia `(i,j)` a unui element al obiectului curent `(a[i][j]` de valoare 1).

Cât timp stiva nu este vidă

{ - extragem un element din stivă pe care îl contorizăm;
- introducem în stivă poziţiile elementelor 1 vecine cu elementul extras
- introducerea în stivă este însoţită de înlocuirea corespunzătoare, în matricea `a[][]`, a lui 1 cu -1.

}

```
#include <iostream.h>
#include<stdlib.h>
const n=6;
int  a[n+2][n+2];

int fig(int i,int j)
  { if(a[i][j]==1)
      { a[i][j]=-1;
        return 1+fig(i-1,j-1)+fig(i-1,j)+fig(i-1,j+1)+
                 fig(i,j-1)+fig(i,j+1)+
                 fig(i+1,j-1)+fig(i+1,j)+fig(i+1,j+1);
      }
    return 0;
  }
int figNR(int i,int j)
 {int *x=new int[n], *y=new int[n], k=0, nr=0;
  x[k]=i; y[k]=j; a[i][j]=-1;
  while(k>=0)
   {i=x[k]; j=y[k];
    k--;  nr++;
    if(a[i-1][j-1]==1)
       { k++; x[k]=i-1; y[k]=j-1; a[i-1][j-1]=-1;};
```

```
    if(a[i-1][j]==1)
       { k++; x[k]=i-1; y[k]=j;   a[i-1][j]=-1;  };
    if(a[i-1][j+1]==1)
       { k++; x[k]=i-1; y[k]=j+1; a[i-1][j+1]=-1;};
    if(a[i][j-1]==1)
       { k++; x[k]=i;   y[k]=j-1; a[i][j-1]=-1;  };
    if(a[i][j+1]==1)
       { k++; x[k]=i;   y[k]=j+1; a[i][j+1]=-1;  };
    if(a[i+1][j-1]==1)
       { k++; x[k]=i+1; y[k]=j-1; a[i+1][j-1]=-1;};
    if(a[i+1][j]==1)
       { k++; x[k]=i+1; y[k]=j;   a[i+1][j]=-1;  };
    if(a[i+1][j+1]==1)
       { k++; x[k]=i+1; y[k]=j+1; a[i+1][j+1]=-1;};
    }
  delete x; delete y;
  return nr;
  }
void generare()
  { int i,j,k,m;
    randomize();
    m=1+random(n*n);
    for(k=1;k<=m;k++)
      { i=1+random(n);
        j=1+random(n);
        a[i][j]=1;
      }
  }
void afisare()
  { int i,j;
    for(i=1;i<=n;i++)
      { for(j=1;j<=n;j++) cout<<' '<<a[i][j];
        cout<<endl;
      }
  }
void main()
 { int i,j,k=0;
   generare();
   afisare();

   cout<<" rezolvare utilizind varianta recursiva"<<endl;
   for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
     if (a[i][j]==1)
      { k++;
        cout<<"fig "<<k<<" = "<<fig(i,j)<<" patrate"<<endl;
      }
   for(i=1;i<=n;i++)
     for(j=1;j<=n;j++)
       a[i][j]=-a[i][j];

   cout<<"rezolvare utilizind varianta nerecursiva"<<endl;
```

```
   k=0;
   for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
     if (a[i][j]==1)
      { k++;
        cout<<"fig "<<k<<" = "<<figNR(i,j)<<" patrate"
              <<endl;
      }
  }
```

I.22. Structuri de date de tip listă

Prin listă înţelegem o succesiune finită de elemente (eventual vidă) aparţinând unei mulţimi de date, ordinea în care apar aceste elemente fiind esenţială. Într-o structură de tip listă ne interesează răspunsurile la următoarele întrebări:

- Care este primul element din listă ?
- Care este ultimul element din listă?
- Care element precede şi care element urmează unui element din listă?

Asupra listelor se efectuează operaţii de tipul:

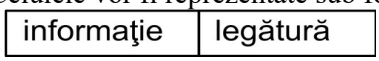
- Accesul la un anumit element din listă, identificat prin valoarea sa sau prin numărul său de ordine în listă;
- Ştergerea (eliminarea) unui element din lista;
- Concatenarea a două sau mai multe liste;
- Descompunerea unei liste în mai multe liste;
- Reordonarea elementelor unei liste conform unui anumit criteriu.

Memorarea listelor se face în două modalităţi:

a) Memorarea cu alocare secvenţială constă în memorarea elementelor listei în locaţii succesive de memorie, conform ordinei acestor elemente în listă: X[1], X[2], ..., X[n], $n \geq 0$. Accesul la elementele listei se face prin intermediul unei variabile de indexare k, $1 \leq k \leq n$.

b)Memorarea înlănţuită presupune că fiecare element al listei este înlocuit cu o celulă (nod) formată din două părţi: o parte de informaţie corespunzătoare elementului şi o parte de legătură ce conţine adresa corespunzătoare următorului element. Ca şi la memorarea secvenţială mai trebuie memorată adresa de bază; în plus partea de legătură a ultimei celule primeşte o valoare ce nu poate desemna o legatură (de obicei valoarea 0).

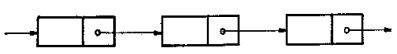
Celulele vor fi reprezentate sub forma



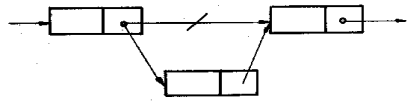
cu semnificaţii evidente.

Legăturile sunt precizate, de obicei, cu ajutorul săgeţilor:

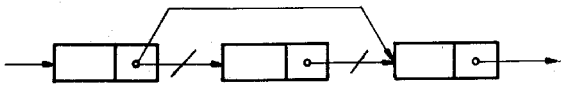
Listă



Inserare



Ştergere



O listă poate fi transformată într-o listă circulară, dacă în partea de legătură a ultimului element vom pune adresa primului element. Alocarea înlănţuită cere mai multă memorie decât cea secvenţială. Trebuie menţionat că în aplicaţii partea de informaţie este mult mai mare decât cea de legătură. Pe de altă parte memorarea înlănţuită elimină necesitatea deplasării de elemente pentru operaţiile de introducere şi scoatere din listă aşa cum rezultă din figura de mai sus.

Implementarea listelor în limbajul C este posibilă deoarece limbajul C permite definirea de tipuri de date care se autoreferă (definite recursiv).

Exemplu:

```
struct nod
   {  int inf;
      nod* leg;
   };
```

Crearea unei liste cu elemente de tipul nod, presupune utilizarea variabilelor cu alocare dinamică (utilizând funcţiile *new* şi *delete*), care se nasc şi dispar în cursul execuţiei programului, la cererea programatorului.

```
void main()
 { nod *p,*q;
   p=new nod;   // alocare
   p->inf=5;
   p->leg=0;

   q=p;           //  q va indica aceeaşi celulă ca şi p

   q=new nod;

   q->inf=p->inf; /* copiază informaţia din celula indicată de p în
celula indicată de q. */

   *q=*p;      // copiază conţinutul celulei pointate de p în celula
pointată de q.

    p->leg=q;  /* face ca în partea de legătură a variabilei desemnate
de p
                                  să se treacă adresa variabilei desemnate de
q. */
```

```
   q->leg=0;     /* face ca partea de legătură a variabilei desemnate
de q
                                 să pointeze către o zonă de memorie nulă */
 }
```

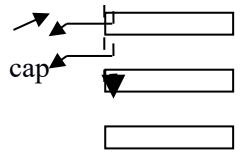
Listele liniare în care inserările, ştergerile şi accesul la valori au loc aproape totdeauna la primul sau la ultimul nod sunt frecvent întâlnite şi poartă denumiri speciale.

I.23. Implementarea unor liste particulare în limbajul C

I.23.1. Stiva

Prin stivă înţelegem o listă liniară pentru care operaţiile de introducere şi extragere a unui element se efectuează la acelaşi capăt al listei, numit vârful stivei. Celălalt capăt al stivei se numeşte baza stivei.

În alocarea înlănţuită, în cazul stivei, legăturile merg de la ultimul element al stivei către primul element, ca în figura următoare:



Următorul program implementează funcţii de lucru cu stiva:

Structura `nod` cuprinde câmpurile unei celule: *inf* – partea de informaţie şi *leg* – partea de legătură.

Funcţia `int inserare(nod *&cap, int x);` realizează inserarea elementului x în stivă. Dacă nu este spaţiu suficient funcţia va returna 0 desemnând eşecul operaţiei de inserare, altfel va returna 1 - inserare cu succes.

Funcţia `void extragere(nod *&cap,int &x);` extrage în x elementul din varful stivei. Apelul are sens dacă stiva este nevidă.

Funcţia `void listare(nod *cap);` afişează elementele stivei.

Funcţia `void stergere(nod *&cap);` şterge stiva.

Funcţia `int virf(nod *cap);` returnează valoarea elementului din vârful stivei.

```
#include<iostream.h>
#include<conio.h>
 struct nod
  {int inf;
   nod *leg;
  };
```

```
int inserare(nod *&cap,int x)
 {
  nod *p=cap;
  cap=new nod; // variabila cap va fi nenulă daca alocarea s-a făcut cu
succes
  if (!cap) return 0;// spatiu insuficient
  cap->inf=x;
  cap->leg=p;
  return 1;
 }

void extragere(nod *&cap,int &x)
 {
  nod *p=cap;
  x=cap->inf;
  cap=cap->leg;
  delete p;
 }
void listare(nod *cap)
 {  while (cap){ cout<<cap->inf<<" "; cap=cap->leg;}  }

int virf(nod *cap){ return cap->inf; }

void stergere(nod *&cap)
 {  nod *p;
    while(cap)
      { p=cap;
        cap=cap->leg;
        delete p;
      }
  }

void main()
{nod *s=0; // Stiva vida
 int op,x;
 do
  {cout<<"1 -Inserare"<<endl;
   cout<<"2 -Extragere"<<endl;
   cout<<"3 -Listare"<<endl;
   cout<<"4 -Virf"<<endl;
   cout<<"5 -Stergere"<<endl;
   cout<<"9 -Stop"<<endl;
   cout<<"Tastati optiunea: ";
   cin>>op;

   switch(op)
    {case 1:cout<<"x=";cin>>x;inserare(s,x);
            break;
     case 2:if (s==0) cout<<"stiva vida"<<endl;
            else {extragere(s,x);
                  cout<<"Am extras elementul "<<x<<endl;
```

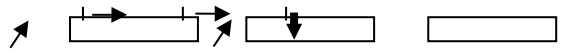
```
                }
              break;
     case 3:cout<<"Continut stiva: ";
              listare(s);
              cout<<endl;
              break;
     case 4:if (s==0) cout<<"Stiva vida"<<endl;
              else cout<<"Virf ="<<virf(s)<<endl;
              break;
     case 5:stergere(s);
    }
  }
  while(op!=9);
}
```

I.23.2. Coada

Prin *coadă* înţelegem o listă în care introducerile se fac la un capăt al listei, numit baza cozii, iar extragerile se fac la celălalt capăt al listei numit vârful cozii.

Alocarea înlănţuită a cozilor presupune cunoaşterea adresei primului element al cozii (vârful cozii) şi a adresei ultimului element al cozii (baza cozii).

În alocarea înlănţuită, legăturile merg ca mai jos:



prim ultim

Următorul program implementează funcţii de lucru cu o listă de tip coadă:

Structura `nod` cuprinde câmpurile unei celule: *inf* – partea de informaţie şi *leg* – partea de legătură.

Funcţia `int inserare(nod *&prim, nod *&ultim, int x);` realizează inserarea elementului x în coadă. Dacă nu este spaţiu suficient funcţia va returna 0 desemnând eşecul operaţiei de inserare, altfel va returna 1 - inserare cu succes.

Funcţia `void extragere(nod *&prim, int &x);` extrage elementul din vârful cozii.

Funcţia `void listare(nod *prim);` afişează elementele cozii.

```
#include<iostream.h>
#include<conio.h>
struct nod
 {int inf;
  nod *leg;
 };
int inserare(nod *&prim,nod *&ultim,int x)
 { if (prim==0)prim=ultim=new nod;
   else ultim=ultim->leg=new nod;
```

```
   if(!ultim) return 0; //esec la alocare
   ultim->inf=x;    ultim->leg=0;
   return 1; // inserare cu succes;
 }
void extragere(nod *&prim,int &x)
 { nod *p=prim;
   x=prim->inf;    prim=prim->leg;
   delete p;
 }
void listare(nod *prim)
 { while(prim)
     {cout<<prim->inf<<" "; prim=prim->leg;  }
 }
int virf(nod *prim){ return prim->inf; }

void stergere(nod *&prim)
 { nod *p;
   while(prim)
    { p=prim;
      prim=prim->leg;
      delete p;
    }
 }

void main()
 { nod *prim=0,*ultim;// coada vida
   int op,x;
   do
    { cout<<"1 -Adaugare"<<endl;
      cout<<"2 -Extragere"<<endl;
      cout<<"3 -Listare"<<endl;
      cout<<"4 -Virf"<<endl;
      cout<<"5 -Stergere"<<endl;
      cout<<"9 -Stop"<<endl;
      cout<<"Tastati optiunea: ";
      cin>>op;
      switch(op)
       { case 1:cout<<"x=";cin>>x;
                inserare(prim,ultim,x);
                break;
         case 2:if (prim==0) cout<<"coada vida"<<endl;
                else { extragere(prim,x);
                       cout<<"Am extras elementul "<<x<<endl;
                     }
                break;
         case 3:cout<<" Coada este:";
                listare(prim);  cout<<endl;
                break;
         case 4:if (prim==0) cout<<"Coada vida"<<endl;
                    else cout<<"Virf ="<<virf(prim)<<endl;
                break;
         case 5:stergere(prim);
```

```
        }
     }
   while(op!=9);
}
```

I.24. Arbori

Prin arbore înţelegem o mulţime finită de elemente, numite noduri $A=\{A_1, A2, ... , A_n\}$, n>0 care are următoarele proprietăţi:

• există un nod şi numai unul numit rădăcina arborelui;

• celelalte noduri (dacă mai există) sunt submulţimi disjuncte ale lui A, care formează fiecare câte un arbore. Arborii respectivi se numesc subarbori ai rădăcinii. Într-un arbore există noduri cărora nu le mai corespund arbori. Un astfel de nod se numeşte nod terminal sau frunză. Un nod rădăcină este numit nod tată, rădăcina unui arbore al său este numită nod fiu iar subarborii sunt descendenţii lui.

O altă noţiune este aceea de nivel. Rădăcina arborelui are nivelul 1.

Dacă un nod are nivelul n, atunci fiii lui au nivelul n+1.

Dacă subarborii unui nod se consideră într-o anumită ordine, subarborele se numeşte ordonat.

I.24.1. Arbori binari

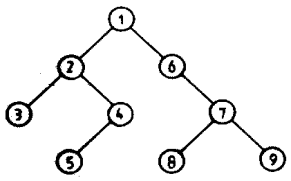
Un arbore binar este o mulţime finită de elemente numite noduri $A=\{A_1,A_2,...,A_n\}$, $n>=0$, care este sau vidă sau conţine un element rădăcină, celelalte elemente împărţindu-se în două submulţimi disjuncte (considerate într-o anumită ordine), care fiecare, la rândul ei, este un arbore binar.

Arborele binar este ordonat, una dintre submulţimi numindu-se subarbore stâng al rădăcinii, iar cealaltă subarbore drept.

Asupra arborilor se pot defini operaţii de tipul:

- construirea unui arbore
- parcurgerea nodurilor
- căutarea unui nod
- inserarea şi ştergerea unui nod
- ştergerea unui arbore

Grafic un arbore binar se reprezintă ca mai jos:



Există trei modalităţi importante de parcurgere a nodurilor unui arbore binar:

• în *preordine*: se vizitează rădăcina, apoi tot în preordine se vizitează nodurile subarborelui stâng şi apoi ale subarborelui drept.

Nodurile arborelui de mai sus, vizitate în preordine, sunt: 1,2,3,4,5,6,7,8,9;

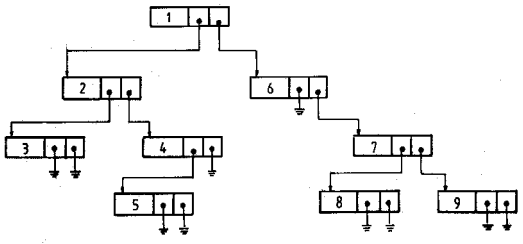
• în *inordine:* se vizitează în inordine nodurile subarborelui stâng apoi se vizitează rădăcina şi apoi se vizitează tot în inordine nodurile subarborelui drept.

Nodurile arborelui vizitate în inordine, sunt: 3,2,5,4,1,6,8,7,9;

• în *postordine*: se vizitează în postordine nodurile subarborelui stâng apoi ale subarborelui drept şi apoi rădăcina.

Nodurile arborelui vizitate în preordine, sunt: 3,5,4,2,8,9,7,6,1.

Un arbore binar se poate implementa uşor cu ajutorul pointerilor, fiecare nod conţinând în afară de informaţia propriu-zisă asociată nodului, adresa fiului stâng şi adresa fiului drept, acestea exprimând legăturile existente între noduri.



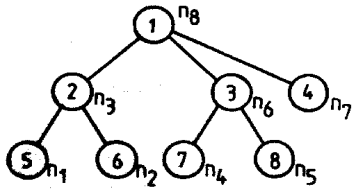

```
#include<iostream.h>
 struct nodArb
   { int inf;
     nodArb *ls,*ld;
   };

void creArb(nodArb *& p)
 { int x;
   cout<<"inf nod(sau punct ptr. sf. ramura):";
   if(cin>>x)
     { p=new nodArb;
       p->inf=x;
       p->ls=0;
       p->ld=0;
       cout<<" nod stanga:";
       creArb(p->ls);
       cout<<" nod dreapta:";
       creArb(p->ld);
     }
   else
     {cin.clear();cin.ignore(80,'\n');}
 }

void prelNod(int x)
 { cout<<x<<" "; }

void Inord(nodArb *p)
  { if(p!=0)
     { Inord(p->ls);
```

```
       prelNod(p->inf);
       Inord(p->ld);
     }
  }

void Preord(nodArb *p)
 { if(p!=0)
    { prelNod(p->inf);
      Preord(p->ls);
      Preord(p->ld);
    }
 }


void Postord(nodArb *p)
 {  if(p!=0)
      { Postord(p->ls);
        Postord(p->ld);
        prelNod(p->inf);
      }
 }

void ActArbSort(nodArb *&p,int x )
  { if(p)
     if (x<p->inf) ActArbSort(p->ls,x);
     else          ActArbSort(p->ld,x);
    else  {p=new nodArb;
           p->inf=x;
           p->ls=0;
           p->ld=0;
          }
  }

void main()
  { nodArb * varf;
    creArb(varf);

    cout<<"Parcurgere in inordine:"<<endl;
    Inord(varf);
    cout<<endl;

    cout<<"Parcurgere in preordine:"<<endl;
    Preord(varf);
    cout<<endl;

    cout<<"Parcurgere in postordine:"<<endl;
    Postord(varf);
    cout<<endl;


    cout<<"Creare arbore de sortare:"<<endl;
    varf=0; // Arbore vid
```

```
    int x;
    cout<<"Tastati un sir de numere terminat cu '.':"<<endl;

    for( ; ; )
      if( cin>>x)ActArbSort(varf,x);
      else break;

    cin.clear();cin.ignore(80,'\n');

    cout<<"sirul sortat:"<<endl;
    Inord(varf);
  }
```

I.24.2. Crearea şi traversarea arborilor oarecare

Pentru implementarea unui arbore oarecare putem folosi următoarea metodă: fiecare nod al arborelui va conţine informaţia propriu-zisă asociată nodului, numărul de descendenţi ai nodului şi adresa fiecărui descendent, acestea exprimând legăturile dintre noduri ca în figura următoare:



Construirea arborelui se face prin citirea în postordine a nodurilor. Pentru fiecare nod se citeşte informaţia utilă şi numărul de descendenţi. Nodurile sunt "păstrate" într-o stivă până când apare nodul ai cărui fii sunt. În acest moment fiii sunt scoşi din stivă şi se actualizează referinţele descendente (de la tată la fii), după care tatăl este pus în stivă. În final singurul nod din stivă va fi rădăcina iar arborele va fi gata construit.

Parcurgerea arborelui se va face pe orizontală (nivel după nivel) utilizând o coadă pentru păstrarea nodurilor ce urmează să fie prelucrate. Iniţial se introduce rădăcina arborelui în coada vidă, după care se scoate pe rând câte un nod din coadă introducându-se descendenţii săi. Se verifică uşor că acest mecanism asigură într-adevăr parcurgerea nivel după nivel.

```
#include<iostream.h>
struct nodA
  { int inf;
    int nrfii;
    nodA* *leg;
  };

struct nod
  { nodA * inf;
```

```
     nod *leg;
   };
void inS(nod *&cap,nodA *x ) // inserare în stivă
 { nod *p=cap;
   cap=new nod;
   if (!cap) {cout<<"Spatiu insuficient" ;return;};
   cap->inf=x;
   cap->leg=p;
 }
void outS(nod *&cap,nodA * &x) //extragere din stivă
 { nod *p=cap;
   x=cap->inf;
   cap=cap->leg;
   delete p;
 }
void inC(nod *&prim,nod *&ultim,nodA* x) // inserare în coadă
 { if (prim==0)prim=ultim=new nod;
   else ultim=ultim->leg=new nod;
   ultim->inf=x;
   ultim->leg=0;
 }
void outC(nod *&prim,nodA* &x) //extragere din coadă
 { nod *p=prim;
   x=prim->inf;
   prim=prim->leg;
   delete p;
 }
void prelNod(int x){cout<<x<<" ";}

void creArb(nodA * &rad)
 { nod *S=0; //pointer la stivă
   int n;
   int inf,nrfii;
   nodA *p;
   cout<<"Nr.total de noduri:"; cin>>n;
   cout<<"Nodurile se vor introduce in postordine"<<endl;
   for(int i=1;i<=n;i++)
     { cout<<"inf. nod "<<i<<":";
       cin>>inf;
       cout<<"nr.fii"; cin>>nrfii;
       p=new nodA;
       p->inf=inf;
       p->nrfii=nrfii;
       if(nrfii>0) p->leg=new nodA*[nrfii];
       for(int k=nrfii;k>=1;k--)
        if(S) outS(S,p->leg[k-1]);
        else{cerr<<"Ati declarat gresit nr. de descendenti!"
                 <<endl;
             return;
            }
        inS(S,p);
      }
```

```
    outS(S,rad);
  }
void traversare(nodA * &rad )
 { if(rad==0)return;
   nod *primC=0,*ultimC=0; //pointeri la coadă
   inC(primC,ultimC,rad);
   nodA* p;
   while(primC)
    { outC(primC,p);
      prelNod(p->inf);
      for(int i=0;i<p->nrfii;i++)
       inC(primC,ultimC,p->leg[i]);
    }
  }
void Preord(nodA* rad)
 { if(rad)
    { prelNod(rad->inf);
      for(int i=0;i<rad->nrfii;i++)
       Preord(rad->leg[i]);
    }
 }
void Postord(nodA* rad)
 { if(rad)
    { for(int i=0;i<rad->nrfii;i++)
       Preord(rad->leg[i]);
      prelNod(rad->inf);
    }
 }
void main()
 { nodA * A;
   creArb(A);
   cout<<"Arborele in preordine:"<<endl;
   Preord(A);
   cout<<endl;
   cout<<"Arborele in postordine:"<<endl;
   Postord(A);
   cout<<endl<<"Arborele pe nivele:"<<endl;
   traversare(A);
   cout<<endl;
 }
```

I.25. Prelucrarea fişierelor în C++

Pentru a realiza operaţii de I/E cu fişiere, trebuie să includem în program fişierul `fstream.h`. În acest fişier sunt definite clasele `ifstream`, `ofstream`, şi `fstream`. Aceste clase sunt derivate din clasele *istream* şi *ostream* derivate la rândul lor din clasa *ios* clase definite în fişierul `iostream.h`. Deci declaraţiile claselor *ios*, *istream* şi *ostream* rămân valabile şi pentru lucru cu fişiere.

În C++, un fişier este deschis prin cuplarea lui la un stream.

Există trei tipuri de streamuri:

- de intrare;
- de ieşire;
- de intrare/ieşire.

Pentru a crea un stream de intrare, el trebuie declarat de tip `ifstream`, iar un stream de ieşire trebuie declarat de tip `ofstream`. Streamurile care realizează ambele tipuri de operaţii vor fi declarate de tip `fstream`.

Constructorii impliciţi ai claselor ifstream, ofstream şi fstream iniţializează streamuri fără a deschide fişiere.

Funcţia open este folosită pentru a asocia un fişier la un stream (după crearea streamului). Această funcţie este membră a tuturor celor trei clase de tip stream şi are prototipul

```
void open( char* numeFişier, int modDeschidere,
           int modProtectie);
```

Definiţii pentru modDeschidere:

Mod deschidere	Valoare	Comentariu
ios::in	0x01	Deschidere pentru citire. Fişierul trebuie să existe. Valoare implicită pentru *ifstream*
ios::out	0x02	Deschidere pentru scriere. Dacă fişierul nu există, se crează iar dacă există, conţinutul său se pierde.
ios::ate	0x04	Deschidere şi poziţionare la sfârşit de fişier.
ios::app	0x08	Deschidere pentru scriere la sfârşit de fişier. Fişierul trebuie să existe.
ios::trunc	0x10	Deschidere şi trunchiere fişier (la lungime 0), dacă există.
ios::nocreate	0x20	Fişierul trebuie să existe la deschidere, altfel se produce eroare.
ios::noreplace	0x40	Fişierul trebuie să fie nou la deschidere, altfel deschiderea eşuează.
ios::binary	0x80	Opusul lui "text" (implicit): nu se traduc perechile "cr/lf".

Observaţie. Pentru activarea mai multor biţi se poate folosi operatorul "sau" binar `(|)`.

Argumentul *modProtectie* poate lua următoarele valori:

0 – fişier normal, fără restricţii de acces (valoare implicită);
1 – fişier de tip "read-only";
2 – fişier ascuns;
4 – fişier de tip sistem;
8 – bit de arhivare activ.

Se pot specifica mai multe dintre atribute folosind operatorul "sau" binar (|).

Exemple:

```
ifstream f;
f.open("f1.dat");
ofstream g;
g.open("f2.dat");
fstream h;
h.open("f3.dat",ios::in|ios::out);
     /* pentru a deschide un  fişier  “fstream",  pentru operaţii de
intrare/ieşire
               trebuie să specificăm atât ios::in cât şi ios::out. */
```

Clasele *ifstream*, *ofstream*, *fstream* conţin funcţii constructor care efectuează în mod automat operaţia de deschidere a fişierului. Ele au aceiaşi parametri şi valori implicite ca şi funcţia open:

Deci putem folosi direct:

```
ifstream f.open("f1.dat");
ofstream g.open("f2.dat");
fstream h("f3.dat",ios::in|ios::out);
```

Dacă operaţia de deschidere eşuează streamul va conţine valoarea zero, ceea ce înseamnă că putem testa uşor dacă operaţia de deschidere a reuşit. Destructorul clasei ifstream (respectiv ofstream sau fstream) goleşte tamponul fişierului şi închide fişierul (dacă nu este deja închis).

Închiderea unui fişier se face cu ajutorul funcţiei membre:

void close(void);

Funcţia membru `int eof()` întoarce o valoare diferită de zero dacă s-a atins sfârşitul fişierului şi 0 în caz contrar.

Scrierea şi citirea într-un, respectiv dintr-un fişier text deschis se poate face utilizând operatorii stream "<<" şi ">>" în mod similar cu cei utilizaţi la operaţiile de I/E utilizând consola cu excepţia faptului că, în loc să utilizăm dispozitivele standard cin, cout etc, folosim streamul cuplat la fişier. Toate informaţiile sunt memorate în fişier în acelaşi format ca şi cel utilizat la afişare.

Când efectuăm operaţii de I/E cu fişiere de tip text, caracterele newline se transformă în combinaţia de caractere `cr/lf`.

Funcţii de I/E de tip binar

Deşi fişierele text sunt utile în foarte multe aplicaţii, ele nu au flexibilitatea fişierelor de tip binar; din acest motiv, limbajul C++ asigură numeroase funcţii de I/E de tip binar: get, put, read, write etc.

Funcţia `get` are mai multe forme şi anume:

```
int get();
```

Extrage din streamul asociat un caracter şi returnează codul caracterului extras.

```
  istream& get(char& c);
```

Citeşte din streamul asociat un caracter şi îl memorează în variabila de ieşire `c`. Funcţia înapoiază o referinţă la streamul asociat care va avea valoarea zero dacă s-a detectat sfârşitul de fişier.

```
  istream& get(char *p,int n, char delimitator='\n');
```

Citeşte n caractere din stream sau până la întâlnirea delimitatorului dat ca al treilea argument şi îi memorează în zona pointată de `p`. Valoarea implicită a delimitatorului este '\n', şi nu este extras din stream.

```
istream& getline(char *p,int n, char delimitator='\n');
```

Este similară cu funcţia precedentă, dar extrage, eventual, şi delimitatorul.

Funcţia put are prototipul:

```
ostream& put (char c);
```

Scrie în streamul asociat caracterul dat ca argument.

Funcţia

```
int peek();
```

ne furnizează următorul caracter din streamul de intrare fără să-l extragă din stream.

Funcţia

```
istream& putback(char c)
```

reinserează un caracter în stream.

Pentru a citi/scrie blocuri de date binare, se utilizează funcţiile read/write, care au următoarele prototipuri:

```
istream& read(unsigned char * p, int n);
ostream& write(const unsigned char *p, int n);
```

Funcţia `read` citeşte `n` octeţi din streamul asociat şi îi plasează în zona de adresă `p`. Dacă se detectează sfărşitul fişierului înainte de citirea tuturor octeţilor, funcţia read opreşte citirea.

Pentru a afla numărul caracterelor citite utilizăm funcţia:

```
int gcount ();
```

Ea înapoiază numărul de caractere citite la ultima operaţie de intrare.

Accesul de tip aleator la fişiere binare

Sistemul de intrare/ieşire din C++ utilizează doi pointeri pentru accesul la fişiere. Primul pointer numit "get pointer" indică poziţia din fişier de unde se va efectua citirea, iar al doilea numit "put pointer" indică poziţia de la care se va face următoarea scriere. Ori de câte ori se va efectua o operaţie de intrare/ieşire pointerul corespunzător va fi avansat (în mod automat) secvenţial.

Accesul aleator la fişiere se poate efectua prin utilizarea funcţiilor `seekg()` şi `seekp()`.

Funcţia

```
ostream& seekg(long n);
```

poziţionează "get pointerul" pe octetul n în fişier (numerotarea se face de la 0).

Funcţia

```
ostream& seekg(long n, seek_dir origine);
```

deplasează "get pointerul" asociat fişierului respectiv cu n octeţi faţă de origine.

`seek_dir` este un tip enumerare cu valorile ios::beg=0, ios::cur=1, ios::end=2 semnificând începutul, poziţia curentă, respectiv sfârşitul streamului.

Funcţiile

```
ostream& seekp(long n);
ostream& seekp(long n, seek_dir origine);
```

sunt similare cu seekg dar se aplică pointerului "put pointer".

I.25.1. Aplicaţii

1) Următorul program conţine funcţii pentru crearea şi exploatarea unui fişier text.

```
#include<fstream.h>
#include<iomanip.h>
struct punct { float x,y;};

void creareFisier()
  { ofstream f("punct.dat");
    punct P; int i=1;
    cout<<"P"<<i<<"(x y):";
    while(cin>>P.x>>P.y) /* citire până la introducerea unui caracter invalid
                                                    pentru tipul numeric */
     { f<<P.x<<" "<<P.y<<endl;   // scriere în fişier
       i++;
       cout<<"P"<<i<<"(x y):";
     }
    cin.clear(); // Anularea indicatorului de eroare
    cin.ignore(80,'\n');
    f.close();
  }
void citireFisier(punct P[],int &n)
  {  ifstream f("punct.dat");
     for(n=0; ;n++)
      { f>>P[n].x>>P[n].y;
        if(f.eof())break;
      }
     f.close();
  }
void afisareVector(float P[],int n)
  {for(int i=0; i<n; i++)
     cout<<"P"<<i<<"("<<P[i].x<<","<<P[I].y<<") ";
   cout<<endl;
  }
void main()
  { punct P[100];int n;
```

```
    creareFisier();
    citireFisier(P,n);
    afisareVector(P,n);
  }
```

2) Următorul program conţine funcţii pentru crearea şi exploatarea unui fişier binar.

```
#include<fstream.h>
#include <iomanip.h>
#include<conio.h>
struct inregistrare
 {char den[20];     // denumire produs
  int zz,ll;        // zi, luna aprovizionare
  float cant,pret;   // cantitate, preţ
 };

void cit(inregistrare& Art)   // citirea unui articol
 {cout<<"denumire:"; cin.get(); //extragerea caracterului<enter>
  cin.get(Art.den,20);
  cout<<"data(zz ll)    :"; cin>>Art.zz>>Art.ll;
  cout<<"cantitate      :"; cin>>Art.cant;
  cout<<"pret           :"; cin>>Art.pret;
 }

void afis(inregistrare Art)
 {cout<<setprecision(2)<<setiosflags(ios::left);
  cout<<setw(20)<<Art.den
      <<setiosflags(ios::right)<<setw(4)<<Art.zz
      <<"/"<<setw(2)<<Art.ll
      <<setw(10)<<Art.cant
      <<setw(10)<<Art.pret;
 }
void actFis(char* denFis)
 { char ok;
   ofstream f(denFis,ios::app|ios::binary);
   inregistrare art;
   while(1)
     { cit(art);
       f.write( (unsigned char*)&art,sizeof(art));
       cout<<"continuati?(R:D/N):"; cin>>ok;
       if(ok=='n'||ok=='N') break;
     }
   f.close();
 }

void creFis(char* denFis) // crează un fişier vid
 {ofstream f(denFis,ios::out|ios::binary);
  f.close();
  actFis(denFis);
```

```
 }
void list(char *denFis) //Listare fişier
 { ifstream f(denFis,ios::in|ios::binary);
   inregistrare art;

   while (1)
    { f.read( (unsigned char*)&art,sizeof(art));
      if(f.eof())break;
      afis(art); cout<<endl;
    }
   f.close();
 }
void sit(char *denFis)
 { ifstream f(denFis,ios::in|ios::binary);
   inregistrare art;
   clrscr();
   float v=0,tvl=0,tvg=0;
   f.read( (unsigned char*)&art,sizeof(art));
   if(f.eof()){cout<<" fisier vid "; return;}
   int luna=art.ll;
   cout<<"  SITUATIA APROVIZIONARII "<<endl;
   cout<<setprecision(2)<<setiosflags(ios::fixed);
   while (! f.eof())
     { if (luna!=art.ll )
        { cout<<"Total luna    "
              <<setw(2)<<luna<<setw(44)<<tvl<<endl;
          tvg+=tvl;
          tvl=0;
          luna=art.ll;
        }
       v=art.cant * art.pret;
       afis(art);
       cout<<setw(10)<<v<<endl;
       tvl+=v;
       f.read( (unsigned char*)&art,sizeof(art));
     }
   cout<<"Total luna "<<setw(2)<<luna
       <<setw(44)<<tvl<<endl;
   tvg+=tvl;
   cout<<"Total general"<<setw(44)<<tvg<<endl;
   f.close();
  }

void main()
  {int op;
   do
    {  cout<<" 1- creare"<<endl;
       cout<<" 2- listare"<<endl;
       cout<<" 3- sit"<<endl;
       cout<<" 4- act"<<endl;
       cout<<" 9- stop"<<endl;
       cout<<"tastati optiunea:";   cin>>op;
```

```
      switch (op)
       { case 1: creFis("fisier.dat");
                 break;
         case 2: list("fisier.dat");
                 break;
         case 3: sit("fisier.dat");
                 break;
         case 4: actFis("fisier.dat");
                 break;
       }
   }
 while (op!=9);
}
```

Bibliografie

1. Barbu Gh., Văduva I., Boloşteanu M., Bazele Informaticii, Editura Tehnică, Bucureşti, 1997.
2. Barbu Gh., Păun V., Calculatoare personale şi programarea în C/C++, Editura Didactică şi Pedagogică, Bucureşti, 2005.
3. Bjarne Stroustrup, The C++ Programming Lanquage, Addison-Wesley Series in Computer Science, 1987
4. Catrina O., Cojocaru I., Turbo C+ +, Editura Teora, Bucureşti, 1993.
5. Costea D., Iniţiere în limbajul C, Editura Teora, Bucureşti, 1996.
6. Dogaru D., Elemente de grafică 3-D, Editura Stiintifică şi Enciclopedică, Bucureşti 1988.
7. Jurcă I., Programarea orientată pe obiecte în limbajul C+ +, Editura Eurobit, Timişoara, 1993.
8. Knuth D. E., Tratat de programarea calculatoarelor. Algoritmi fundamentali, Editura Tehnică, Bucureşti, 1976.
9. Livovschi L., Georgescu H., Sinteza şi analiza algoritmilor, Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1986.
10. Logofătu D., Bazele programării în C, Editura Polirom, Bucureşti, 2006.
11. Marinoiu C., Programarea în limbajul C, Editura Universităţii din Ploieşti, 2000.
12. Miroiu M., Traşcă I., Tablouri bidimensionale în C/C++ şi Matlab, Editura Hoffman, 2010.
13. Muşlea I., Iniţiere în C++. Programarea orientată pe obiecte, Editura MicroInformatica, Cluj-Napoca, 1993.
14. Negrescu L., Limbajul Turbo C, Editura MicroInformatica, Cluj-Napoca, 1993.
15. Oprea N., Programare orientată pe obiecte, Editura Matrix, Bucureşti, 2003.
16. Păun V., Algoritmică şi programarea calculatoarelor. Limbajul C+ +, Editura Universităţii din Piteşti, 2003.
17. Somonea D., Turturea D.,Iniţiere în C++, Editura Tehnică, Bucureşti, 1996.
18. Stoilescu D., Manual de C/C++ pentru licee, Editura Radial, Galaţi, 2004.
19. Tudor S., Tehnici de programare şi structuri de date, Editura L&S Info-Mat, Bucureşti, 1994.
20. Zaharia D. M., Structuri de date şi algoritmi în limbajele C şi C++, Editura Albastră, Cluj Napoca, 2002.
21. ***, Colecţia Gazeta de informatică