

Sergiu G. Istrati

P R O G R A M A R E

Inițializare în limbajele C și C++

Ciclu de prelegeri la disciplina “Programare”



Chișinău 2003

Adnotare

Lucrarea de față este destinată studenților anilor I și II secțiile de învățământ de zi și fără frecvență a Universității Tehnice a Moldovei ce studiază disciplina Programare și în special studenților Facultății de Radioelectroncă și Telecomunicații catedra Sisteme Optoelectronice cu specializările 1871 Inginerie și Management în Telecomunicații și 2105 Sisteme Optoelectronice.

Autor: lector superior Sergiu G.Istrati

Redactor responsabil: conf. univ. dr. Pavel Nistiriuc

Recenzent: academician, dr. hab. Victor I. Borșevici

U.T.M. 2003

Cuprins

Limbajul de programare C/C++

Întroducere	5
1. Alfabetul limbajului	6
2. Structura unui program	8
3. Tipuri de date	12
3.1. Tipuri de date simple predefinite. Constante	12
3.1.1. Constante întregi	12
3.1.2. Constante reale	14
3.1.2. Constante character	15
3.1.3. Șiruri de caractere	16
4. Variabile	16
4.1. Nume de variabile (identificatori)	16
4.2. Descrierea variabilelor	16
4.3. Inițializarea variabilelor	17
5. Operații și expresii	18
5.1. Operații aritmetice	18
5.2. Operația de atribuire	18
5.3. Operații de incrementare(decrementare)	18
5.4. Relații și operații logice	19
5.5. Operațiile logice poziționale	20
5.6. Operația dimensiune	22
5.7. Operația virgulă	22
5.8. Expresii condiționate	23
5.9. Conversii de tip	23
5.10. Prioritățile operațiilor	25
6. Instrucțiuni	26
6.1. Tipurile instrucțiunilor	26
6.2. Instrucțiuni expresie	27
6.3. Instrucțiuni de ramificare (condiționale)	28
6.3.1. Instrucțiunea de ramificare IF și IF-ELSE	28
6.3.2. Instrucțiunea de salt necondiționat GOTO	29
6.3.3. Instrucțiunea de selectare SWITCH	30
6.3.4. Instrucțiunea de întrerupere BREAK	32
6.4. Instrucțiuni iterative(ciclice)	33
6.4.1. Instrucțiunea ciclică FOR	33

6.4.2. Instrucțiunea ciclică WHILE	34
6.4.3. Instrucțiunea de ciclare DO_WHILE	35
6.4.4. Instrucțiunea de continuare CONTINUE	36
7. Masive	36
7.1. Descrierea masivelor	36
7.2. Accesul la elementele masivului	37
7.3. Inițializarea masivelor	38
7.4. Exemple de prelucrare a masivelor	40
8. Șiruri de caractere	41
8.1. Masive de șiruri	43
9. Structuri în C/C++	44
9.1. Declararea variabilelor de tip structură	45
9.2. Inițierea variabilelor tip înregistrare	45
9.3. Folosirea structurilor	46
9.4. Structuri imbricate	47
9.5. Masive de structuri	47
10. Funcții în C/C++	49
10.1. Transmiterea parametrilor în funcție	52
10.2. Întoarcerea valorilor din funcție	54
10.3. Prototipul funcției	56
10.4. Variabile locale și domeniul de vizibilitate	57
10.5. Variabile globale	59
10.6. Conflicte dintre variabile locale și globale	60
11. Indicatori (pointeri)	61
11.1. Indicatori și funcții	64
12. Fișiere în C/C++	68
12.1. Deschiderea fișierelor	70
12.2. Funcții de înscriere/citire din fișier	73
12.2.1. Înscriere/citire de caractere	74
12.2.2. Înscriere/citire de șiruri	75
12.2.3. Întrare/ieșire cu format	77
12.2.4. Fișiere și structuri	78
Anexa1. Funcții de intrare-ieșire în C/C++	81
Anexa 2. Funcții matematice	89
Anexa 3. Funcții folosite la prelucrarea șirurilor de caractere	97

Întroducere

Scopul prezentei lucrări este familiarizarea studenților cu principalele instrumente și metode de programare în limbajele C și C++.

Limbajul de programare C a fost elaborat de către Denis M.Ritchi în 1972 și descris detaliat în cartea “Limbajul de programare C” de Ritchi și Brian B.Kernigan. Realizarea limbajului în conformitate cu regulile descrise în carte poartă denumirea de “Standard C K&R” și este realizarea standard minimală. În 1983 a fost creat un nou standard C de către American National Standards Institute numit “Standard ANSI-C”. Mai apoi a fost elaborat limbajul C++ ca o derivată a limbajului C.

În așa fel limbajul C++ posedă marea majoritate a posibilităților limbajului ANSI-C și în plus la acestea alte instrumente de programare cu posibilități mai avansate.

Lucrarea de față conține descrierea atât a instrumentelor din limbajul ANSI-C ce sînt susținute și de compilatoarele C++, cît și descrierea instrumentelor de programare ale limbajului C++.

Deasemenea în lucrare este atrasă o deosebită atenție exemplelor practice de rezolvare a diferitor tipuri de probleme cu lămurire detaliată.

Prezenta lucrare este o parte din suita de lucrări didactico- metodice elaborate de către lectorul superior Sergiu G. Istrati îndreptate spre optimizarea procesului de instruire a studenților la disciplina Programare. Au fost elaborate următoarele lucrări:

- Ciclu de prelegeri la disciplina Programare. Limbajul Pascal.
- Ciclu de prelegeri la disciplina Programare. Limbajul C. (prezenta lucrare)
- Îndrumar metodic privind îndeplinirea lucrărilor de laborator.
- Îndrumar metodic privind îndeplinirea lucrărilor individuale.
- Îndrumar metodic privind îndeplinirea lucrării de curs.

Toate aceste lucrări pot fi accesate în Internet pe adresa www.istrati.com

Limbajul de programare C.

1. Alfabetul limbajului.

Numim limbaj de programare un limbaj prin care putem comunica unui calculator metoda de rezolvare a unei probleme. Iar metoda de rezolvare a problemei, după cum știm deja o numim algoritm. Întreaga teorie informatică se ocupă de fapt cu elaborarea unor noi calculatoare, limbaje de programare și algoritmi. Datoria oricărui limbaj de nivel înalt este să ne pună la dispoziție o sintaxă cât mai comodă prin care să putem descrie datele cu care lucrează programul nostru și instrucțiunile care trebuie executate pentru a rezolva o anumită problemă.

Limbajul de programare C ca și orice alt limbaj de programare își are alfabetul său și specificul de utilizare a simbolurilor. Alfabet al unui limbaj de programare se numește un set de simboluri permis pentru utilizare și recunoscut de compilator, cu ajutorul căruia pot fi formate mărimi, expresii și operatori ai acestui limbaj de programare. Alfabetul oricărui limbaj de programare conține cele mai simple elemente cu semnificație lingvistică, iar sintaxa limbajului definește modul în care se combină elementele vocabularului pentru a obține fraze corecte (instrucțiuni, secvențe de instrucțiuni, declarații de tipuri, variabile, constante, etichete, funcții, proceduri etc.). Elementele vocabularului sunt alcătuite din caractere. Orice caracter este reprezentat în calculator, în mod unic, printr-un număr natural cuprins între 0 și 127, numit cod ASCII. Mulțimea elementelor alfabetului limbajului de programare C se poate împărți în 5 grupe:

1) Simboluri folosite la formarea identificatorilor și cuvintelor cheie. În componența acestei grupe intră literele minuscule și majuscule ale alfabetului latin (englez) și simbolul subliniere “_”. Este de menționat faptul că literele minuscule și majuscule de același fel (Exemplu: a și A) sunt interpretate ca simboluri diferite din cauza că au diferite coduri ASCII.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z _

2) Literele minuscule și majuscule ale alfabetului rus (sau altui alfabet național) și cifrele arabe.

А Б В Г Д Е Ж З И К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я

а б в г д е ж з и к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я

0 1 2 3 4 5 6 7 8 9

3) Simboluri speciale ce se folosesc la organizarea proceselor de calcul și la transmiterea compilatorului unui set de instrucțiuni.

Simbolul	Denumirea	Simbolul	Denumirea
,	Virgulă)	Paranteză rotundă închisă
.	Punct	(Paranteză rotundă deschisă
;	Punct-virgulă	}	Paranteză figurată închisă
:	Două puncte	{	Paranteză figurată deschisă
?	Semnul întrebării	<	Mai mic
'	Apostrof	>	Mai mare
!	Semnul exclamării	[Paranteză patrată deschisă
	Linie verticală]	Paranteză patrată închisă
/	Slash	#	Număr (diez)
\	Slash inversat	%	Procent
~	Tilda	&	Ampersand
*	Steluța	^	Negare logică
+	Plus	=	Egal
-	Minus	"	Ghilimele

4) Simboluri de conducere și de despărțire. În componența acestei grupe intră spațiul (blank-ul), simbolul tabulare, simbolul de trecere în rând nou, întoarcerea căruciorului, linie nouă și pagină nouă. Aceste simboluri au destinația de a despărți obiectele determinate de utilizator așa ca constante și identificatori. O consecutivitate de simboluri de despărțire este precautată de către compilator ca un singur simbol. (Exemplu: mai multe blank-uri consecutive).

5) Pe lângă grupele de simboluri precautate limbajul C pe larg folosește consecutivități de conducere, adică combinații de simboluri speciale folosite în funcțiile de intrare-ieșire a informației. Consecutivitatea de conducere este alcătuită dintr-un slash inversat (\), care se află neapărat pe primul loc, după care urmează o combinație din litere latine și cifre.

Consecutivitatea de conducere	Denumirea	Echivalentul hexazecimal
\a	Sunet (beep)	007
\b	Întoarcere cu o poziție	008
\t	Tabulare orizontală	009
\n	Trecere în rând nou	00A
\v	Tabulare verticală	00B

\r	Întoarcerea căruciorului	00C
\f	Trecerea în alt format	00D
\"	Ghilimele	022
\'	Apostrofa	027
\0	Simbolul zero	000
\\	Slash inversat	05C
\ddd (d-cifră)	Simbolul grupului de coduri PC în sistemul octal	
\xddd (d-cifră)	Simbolul grupului de coduri PC în sistemul hexazecimal	

Consecutivitatea de tipul \ddd și \xddd (aici prin d este notată o cifră oarecare) permite a scrie un cod al calculatorului ca o consecutivitate de cifre octale sau hexazecimale respectiv. De exemplu simbolul de întoarcere a căruciorului poate fi interpretat în diferite moduri: \r – consecutivitatea generală de conducere, \015 - consecutivitatea octală de conducere, \x00D - consecutivitatea hexazecimală de conducere.

În afară de aceasta în limbaj sunt rezervate cuvinte de serviciu, numite cuvinte cheie care pot fi folosite într-un sens strict definit: *int, float, double, char, long, signed, unsigned, const, volatile, sizeof, if, else, goto, case, default, for, while, do, break, continue, near, far, void, return, pascal, cdecl, interrupt, auto, extern, static, register, union, enum, typedef, asm, _cs, _ds, _es, _ss, _AH, _AX, _BX, _BL, _CH, _CL, _CX, _DX, _DL, _BP, _DI, _SI, _SP*.

Cuvintele cheie definesc sensul semantic al instrucțiunilor din C. Cuvintele cheie ce încep cu semnul _ (subliniere) se folosesc pentru acces la segmentele de date și la registrele calculatorului.

Prin sintaxa unui limbaj de programare se înțelege, în general, un ansamblu de reguli de agregare a unităților lexicale pentru a forma structuri mai complexe (instrucțiuni, declarații, programe etc.) Structura unui program în limbajul de programare C deasemenea își are regulile ei de sintaxă după cum urmează: un antet, urmat de funcțiile secundare după care urmează corpul funcției principale (care conține în componența sa și o parte declarativă). Și pentru descrierea în detaliu a acestor componente sunt necesare, desigur și alte reguli.

2. Structura unui program.

Pentru a primi o reprezentare generală a programului în C să urmărim un exemplu concret. Presupunem, că există un masiv unidimensional x cu n elemente de

tip întreg și e necesar să alcătuim un program, care calculează și afișază suma elementelor întregului masiv. Aceasta este o problemă tipică pentru prelucrarea masivelor unidimensionale.

În concepțiile limbajului C fiecare algoritm evidențiat se realizează de o unitate de program numită funcție. Stilul de programare în C se caracterizează prin tendința de a evidenția un număr mare de funcții nu prea voluminoase, astfel, ca prelucrarea datelor în aceste funcții să nu depindă de celelalte părți a programului. Acest lucru face programul destul de înțeles și dă posibilitatea de a introduce ușor corecții în unele funcții fără a tangenta celelalte. Prin funcțiile evidențiate unica cu care începe îndeplinirea programului se numește principală și are denumirea fixată: main. Toate celelalte funcții au denumire arbitrară, pe care programatorul singur le numește. Ele pot fi înscrise în fișierul inițial pînă la funcția main (în ordine arbitrară) sau se pot afla în fișiere diferite pe un suport magnetic. În programul de mai jos, ce realizează rezolvarea problemei intră funcția main() și funcția numită suma(). Să analizăm acest program:

```
#include<stdio.h>
#include<conio.h>
int suma (int y[10], int m)  {
    int i, suma=0;
    for (i=0;i<m;i++) {
        suma +=y[i]; } return (suma); }
void main (void) {
    int w,n,i,x[10]; clrscr();
    printf("Culege marimea masivului n<10\n");
    scanf("%d", &n);
    printf("Culege masivul x[%d]\n",n);
    for (i=0;i<n;i++) {
        printf("Culege elementul x[%d]\n",i);
        scanf("%d",&x[i]); }
    w=suma(x,n);
    printf("suma=%d\n",w);
    getch(); }
```

Primele două rînduri: `#include <stdio.h>` și `#include <conio.h>` nu sînt instrucțiuni ale limbajului C. Simbolul “#” indică că acestea sînt directive ale procesorului. Procesorul execută prelucrarea prealabilă a textului programului înainte de compilare. În cazul dat, aceste directive ne dau de înțeles că în fișierul ce se compilează trebuie introdusă informația din fișierele sistemului Turbo C `stdio.h`

(Standard Input/Output Header – titlu de introducere-extragere) și *conio.h* (Console Input / Output Header- titlu de introducere-extragere de la consolă). Existența acestor directive este condiționată de faptul că în textul programului sînt folosite funcțiile încorporate *printf()* și *clrscr()*, informația despre care se conține în fișierele indicate. Următorul rînd „*int suma (int y[10], int m)*” conține declararea funcției *suma()* cu 2 parametri: un masiv *y[10]* și o variabilă simplă *m* de tip întreg. Aici e necesar de menționat faptul, că în limbajul C orice program începe de la funcția principală *main()*, independent de faptul cîte funcții auxiliare se conțin în program. Luînd în considerație acest fapt, funcția *suma()* va fi analizată la momentul cînd apelul ei va fi efectuat din corpul funcției principale *main()*.

Rîndul din program: *void main (void)* definește titlul funcției principale cu numele *main()*. Cuvîntul *void* din fața funcției semnifică faptul, că această funcție nu va întoarce valori în procesul execuției sale. Parantezele rotunde ce urmează după *main()* indică compilatorului că aceasta este o funcție, iar cuvîntul *void* din paranteze – faptul că funcția nu folosește parametri. Perechea acoladelor: prima ce se deschide după *main()* și corespunzător acolada ce se închide după funcția *getch()*; – mărginește instrucțiunile care formează corpul funcției principale *main()*. În limbajul C perechea de accolade *{}* mărginește o secvență de instrucțiuni care se precaută ca un tot întreg. Următoarul rînd conține descrierea variabilelor folosite în funcția principală *main()*: *int w,n,i,x[10];* care transmite compilatorului că în program vor fi folosite variabilele *w,n* și *i* de tip întreg și un masiv *x* cu mărimea de 10 elemente de tip întreg. După descrierea datelor urmează instrucțiunea de adresare la funcția *clrscr()* încorporată în biblioteca *conio.h* din Turbo C.

Această funcție are destinația de curățire a ecranului. După ea urmează funcția *printf()*, care afișază pe ecran un comentariu. În acest caz funcția *printf("Culege marimea masivului n<10\n");* afișază pe ecran propunerea de a culege mărimea masivului și simbolul *\n* de trecere în alt rînd. Funcția următoare *scanf("%d", &n);* este o funcție de intrare și face posibilă introducerea de la tastatură a valorii mării masivului *n*. Simbolul *%d* indică funcției că valoarea citită este de tip întreg, iar simbolul *&* indică adresa de memorie unde va fi înscrisă valoarea lui *n*. Funcția *printf("Culege masivul x[%d]\n",n);* deasemenea afișază un comentariu pe ecran și propune culegerea valorilor elementelor masivului *x* cu mărimea *n* deja cunoscută. Următoarea instrucțiune este instrucțiunea ciclică *for*. Această instrucțiune este compusă și are ca sarcină repetarea unui set de instrucțiuni de cîteva ori cu diferite valori ale parametrilor.

```
for (i=0;i<n;i++) {printf("Culege elementul x[%d]\n",i); scanf("%d",&x[i]);}
```

aici cuvîntul *for* este cuvînt rezervat, *i*-parametrul ciclului, care-și schimbă valoarea de la 0 la *n* cu pasul 1 datorită instrucțiunii de incrementare *i++*. Corpul ciclului, care va fi repetat de *n* ori este mărginit de perechea de accolade deschisă și închisă, și este compus din 2 funcții: prima *printf()*, care afișază un comentariu pentru culegerea valorii elementului current al masivului și a doua *scanf()*, care face posibilă înscrierea valorii elementului current al masivului de la tastatură în memorie. În așa fel, la sfîrșitul îndeplinirii ciclului toate elementele masivului *x[n]* vor avea valori, fapt ce face posibilă calcularea sumei totale a elementelor din masiv. După executarea instrucțiunii ciclice *for* urmează instrucțiunea de atribuire *w=suma(x,n)*; . În partea dreaptă a acestei atribuirii este amplasată funcția *suma()*. Asume valoarea acestei funcții, după executarea cu folosirea parametrilor locali *x* și *n*, va fi atribuită variabilei *w*. Să analizăm funcția *suma()*:

```
int suma (int y[10], int m) {    int i, suma=0;
for (i=0;i<m;i++) { suma +=y[i]; } return (suma); }
```

Ca și la declararea oricărei funcții în limbajul C întîi de toate urmează antetul funcției: *int suma (int y[10], int m)* , unde cuvîntul *int* din fața numelui funcției *suma* este tipul valorii întors de funcție în programul principal *main()*. Parantezele rotunde după numele funcției mărginesc lista de parametri formali folosiți de funcție. Acești parametri sînt un masiv de tip întreg *y* cu lungimea 10 elemente și o variabilă de tip întreg *m*. Este de menționat faptul că la momentul chemării funcției *suma()* din cadrul programului principal *main()*, valorile parametrilor actuali (în cazul nostru *x*-masivul prelucrat și *n*-mărimea lui) sînt atribuite parametrilor formali din funcție (în cazul nostru masivul *y* și mărimea lui *m*) și este necesară îndeplinirea următoarei condiții: parametrii actuali trebuie să corespundă ca cantitate, poziție și tip cu parametrii formali. Acoladele deschisă după antetul funcției și corespunzător închisă după operatorul *return()*; delimitează corpul funcției *suma()*, care își are și ea secțiunea de declarare a datelor. Aici *int i, suma=0*; declară variabila locală *i* de tip întreg și inițializează funcția *suma()* cu valoarea 0. Următoarea este instrucțiunea ciclică *for* care conține în corpul său delimitat de accolade o singură instrucțiune: instrucțiunea compusă de atribuire *suma+=y[i]*; echivalentă cu instrucțiunea *suma=suma+y[i]*; care calculează suma elementelor din masiv. Ciclul va fi repetat de *m* ori cu diferite valori a parametrului *i*, unde *m* este mărimea masivului, adică cantitatea elementelor din masiv, iar *i*-numărul de ordine a elementului current din masiv. După executarea ciclului variabila *suma* va conține valoarea finală a sumei tuturor elementelor din masiv. Transmiterea acestei valori funcției principale *main()* este efectuată de către operatorul *return(suma)*; . După executarea acestui operator valoarea sumei va fi

inclusă în locul de unde a fost chemată funcția *suma()*, în cazul nostru aceasta este instrucțiunea de atribuire $w=suma(x,n)$;

Deci, valoarea sumei elementelor masivului va fi atribuită variabile *w*. După aceasta urmează afișarea pe ecran a rezultatului final: `printf("suma=%d\n",w);` . Ultima instrucțiune din program este apelul la funcția *getch()*, care oprește executarea programului cu scopul vizualizării rezultatului până când nu va fi culeasă tasta *Enter*.

În așa fel poate fi descrisă structura generală a unui program în C după cum urmează: orice program începe cu includerea bibliotecilor de funcții care vor fi folosite în program, după aceasta urmează declararea tuturor funcțiilor auxiliare folosite în program, care au următoarea componentă: antetul funcției, secția de declarare a variabilelor, constantelor locale, după care urmează corpul funcției; după declararea tuturor funcțiilor auxiliare urmează corpul funcției principale *main()* delimitat de o pereche de accolade, care conține descrierea variabilelor, constantelor și însăși instrucțiunile programului principal.

3. Tipuri de date.

Un program în limbajul C conține o descriere a acțiunilor ce trebuie să fie executate de calculator și o descriere a datelor ce sunt prelucrate de aceste acțiuni. Acțiunile sînt descrise prin instrucțiuni, iar datele prin declarații (sau definiții). Prin tip de date se înțelege o mulțime de valori care pot fi atribuite unei variabile sau constante. Tipurile de date în C pot fi împărțite în două categorii: simple (elementare) și compuse (structurate). În general, tipurile de date sunt definite explicit de către programator și sunt specifice programului în care apar. Există însă tipuri de date elementare de interes mai general, numite tipuri predefinite a căror definiție se consideră cunoscută și nu cade în sarcina programatorului.

3.1. Tipuri de date simple predefinite. Constante.

Un program în C conține în mod explicit diferite valori textuale și numerice. Așa valori, ce apar în program, sînt numite constante. Constanta – este o valoare numerică sau textuală, care întotdeauna este definită și în mersul îndeplinirii programului rămîne neschimbată. Tipul constantei se definește de forma ei de înscris, iar valoarea ei este încheiată în ea înseși.

3.1.1. Constante întregi.

Constanta întreagă este un număr înscris în program fără punct zecimal și fără indicatorul de putere. Constantele întregi în C pot fi: Zecimale, Octale, Hexazecimale.

Sistemul de enumerare a constantelor este recunoscut de compilator după forma lor de înscriere. Dacă constanta este înscrisă prin intermediul cifrelor 0..9 și prima cifră nu e zero, atunci constanta se consideră zecimală. De exemplu: 123, 45, 37. Dacă constanta este înscrisă folosind cifrele 0..7 și prima cifră este zero, atunci constanta se consideră octală. De exemplu: 045, 037. Dacă constanta este înscrisă cu ajutorul cifrelor 0..9 și literelor a..f sau A..F și se începe cu 0x sau 0X, atunci constanta este hexazecimală. De exemplu: 0x45, 0x37. În aceste exemple constantele înscrise cu unele și aceleași cifre au valori diferite, ce se definesc de baza sistemului de enumerare.

Pentru determinarea constantelor de tip întreg sunt folosite diferite cuvinte rezervate, care determină diapazonul valorilor și volumul de memorie rezervat pentru constantă.

Tipul	Volumul de memorie (octeți)	Diapazonul de valori
int	2	-32768 ... 32767
Short (short int)	2	0 ... 255
Long (long int)	4	-2 147 483 648 ... 2 147 483 647
unsigned int	2	0 ... 65 535
unsigned long	4	0 ... 4 294 967 295

În dependență de valoarea constantei, compilatorul alocă pentru reprezentarea ei în calculator doi sau patru octeți de memorie. Pentru valorile -32768...32767 se alocă doi octeți, unde primul bit se interpretează ca semnul constantei, iar 15 biți rămași definesc valoarea ei. În așa caz constanta este de tipul *int* (întreg). Pentru valorile de la 0 ... 65535 se alocă doi octeți de memorie, însă toți 16 biți definesc valoarea constantei. Așa fel de constantă are tipul *unsigned int* (întreg fără semn). Constantele acestui diapazon, înscrise cu semnul minus se cercetează ca fără semn, la care se aplică operația "minus unar".

Pentru valorile de la -2 147 483 648 pîna la 2 147 483 647 se alocă 4 octeți, la care primul bit se interpretează ca semn, iar 31 de biți rămași – ca valoare a numărului. Așa constante au tipul *long* (lung).

Pentru valorile de la 0 pîna la 4 294 967 295 se alocă 4 octeți, la care toți 32 de biți se interpretează ca valoare. Așa constantă are tipul *unsigned long* (lung fără semn). Pentru acest tip la constantele negative se aplică operația "minus unar". Analogic se alocă memorie pentru constantele octale și hexazecimale, aflate în diapazonul zecimal respectiv.

Constantele, ce sînt înscrise în program sub forma unui număr mai mare ca 4 294 967 295 aduc la supraîncărcare, însă compilatorul nu face preîntîmpinare, iar în memorie se înscriu biții inferiori ai constantei trunchiate. Programatorul are posibilitate de a indica explicit compilatorului, că pentru o oarecare constantă este necesar de alocat 4 octeți și de a indica interpretarea lor ca fără semn (*unsigned*). Pentru aceasta se folosesc modificatori speciali, înscriși după cifra mai mică a constantei. Pentru a indica, că constanta are tipul long, trebuie de înscris modificatorul L sau I (se permite de asemenea și l sau i).

În standardul K&R C inițializarea constantelor de tipul întreg are următoarea sintaxă:

#define name value care este plasată pînă la funcția *main()*. Unde *#define* este directiva compilatorului, *name* – numele constantei, *value* – valoarea constantei. De exemplu: *#define K 35* sau *#define salariu 700*. Între directivă, nume și valoarea constantei este necesară prezența la minim un spațiu. În exemplele de mai sus compilatorul implicit va atribui variabilelor *K* și *salariu* valori de tipul întreg, întrucît numerele 35 și 700 au sintaxa și formatul constantelor de tip întreg.

Unele compilatoare C, ce susțin standardul ANSI-C permit inițializarea constantelor în 2 moduri. Primul mod cu folosirea directivei *#define* a fost descris mai sus. Al doilea mod folosește cuvîntul rezervat *const* pentru inițializarea constantei, descrierea tipului ei și atribuirea valorii și are următoarea sintaxă:

const int name = value; care este plasată după funcția *main()*, unde *const* este cuvînt rezervat pentru inițializarea constantelor, *int* cuvîntul rezervat pentru desemnarea constantelor de tip întreg și *value* – valoarea constantei. De exemplu:

```
main() {  
    const int K = 35;  
    ... ;  
    const int salariu = 700;  
    ... ; }
```

3.1.2. Constante reale.

Constantele reale (flotante) reprezintă numere zecimale fracționare ce pot fi scrise sub două forme: formă cu punct și formă cu ordin. Constanta reală în formă cu punct se scrie ca fracție zecimală cu semn sau fără semn, la care partea întreagă și partea fracționară se despart prin punct. Dacă semnul constantei este omis ea se socrate pozitivă. Constanta reală în formă cu ordin este comodă pentru scrierea numerelor foarte mari sau foarte mici. În C, ca și în majoritatea limbajelor de programare, pentru așa înscrieri se folosesc constante reale în formă cu ordin, ce au

aspectul: *mantisa_e_ordinul* sau *mantisa_E_ordinul*. În aceste notații în calitate de mantisă poate fi scrisă sau o constantă zecimală fără modificator sau o constantă reală în formă cu punct. În calitate de ordin se scrie o constantă întreagă zecimală, posibil cu semn, ce determină puterea numărului zece. Dacă semnul ordinului lipsește se subînțelege semnul +. De exemplu $7.32E+14 = 7.32 \cdot 10^{14}$; $55.000000E-3 = 0.055$;

Pentru determinarea constantelor de tip real sunt folosite diferite cuvinte rezervate, care determină diapazonul valorilor și volumul de memorie rezervat pentru constantă.

Tipul	Volumul de memorie (octeți)	Diapazonul de valori
Float	4	3.4E-38 ... 3.4E+38
Double	8	1.7E-308 ... 1.7E+308
Long double	10	3.4E-4932 ... 3.4E+4932

Constantele reale au un nivel oarecare de aproximație, care depinde de compilator. În așa fel cifra 6.12345678912345 în diapazonul de aproximație pentru tipul *float* va fi interpretată de compilator ca 6.123456, acest tip se mai numește tip cu aproximație unitară și are aproximația –6 poziții după punct. Tipul *double* se mai numește tip cu aproximație dublă și are aproximația de până la 15 –16 poziții după punct. Sintaxa de inițializare a constantelor de tip flotant (real) este următoarea:

După standardul K&R-C : *#define PI 3.14*.

După standardul ANSI-C *const float PI = 3.14*; inclusiv cel din K&R C.

3.1.3. Constante caracter.

O constantă caracter este un oarecare caracter al alfabetului luat în apostrofe. Exemplu: *'caracter'*.

Valoarea unei constrante de tip caracter (char) poate fi o literă, o cifră sau un alt simbol al tastaturii. Pentru fiecare constantă de tip caracter în memorie se alocă câte un octet. Mulțimea valorilor unei constante de tip carecter este următoarea: literele minuscule și majuscule ale alfabetului latin, zece cifre arabe 0..9 și simboluri speciale ! @ # \$ % ^ & * () _ + = | \ / } { " ' : ; ? > . < , ~ `

Există doua metode de scriere a caracterelor.

Prima metodă: Orice caracter al tabelului codurilor ASCII poate fi reprezentat în formă de constanta caracter astfel: *'ddd'* sau *'\xHHH'*, unde *ddd*–codul octal, *HHH*–codul hexazecimal al caracterului. Zerourile ce stau în fața codului caracterului pot fi omise.

Metoda a doua: Caracterele des folosite ce nu se reflectă pe ecran nu trebuie scrise sub formă de cod. În acest caz se folosesc notațiile lor rezervate. Aceste caractere în C sunt repartizate în clasa caracterelor de control. Dacă caractere de control se întâlnesc, spre exemplu, în rîndul de extragere, atunci ele provoacă o acțiune corespunzătoare.

Sintaxa de inițializare a constantelor de tip caracter (char) este următoarea:

După standardul K&R-C : *#define Lit 'C'*.

După standardul ANSI-C *const char Lit = 'C'*; inclusiv cel din K&R C.

3.1.4. Șiruri de caractere.

Un șir este o succesiune de caractere ale alfabetului cuprinse între ghilimele. Spre deosebire de alte limbaje de programare, limbajul C nu conține un tip de date special ce desemnează șirurile de caractere. Limbajul C operează cu șirurile cum ar lucra cu o succesiune de date de tip caracter amplasate într-o structură numită masiv. Aici fiecare simbol din șir este o componentă aparte a masivului de tip char. Tipul masiv este un tip structurat în C și va fi studiat în capitolul “Tipuri structurate”

Notă: Unele compilatoare C și C++ susțin tipuri de date speciale pentru operare cu șiruri de caractere și conțin biblioteci de funcții pentru prelucrarea șirurilor.

4. Variabile.

Variabila este o mărime care în procesul îndeplinirii programului poate primi valori diferite. Pentru variabile programatorul trebuie să determine notații caracteristice proprii care se numesc identificatori. Deseori identificatorii sunt numiți nume simbolice sau pur și simplu nume.

4.1. Nume de variabile (identificatori).

Numele variabilei (identificatorul variabilei) este o succesiune de caractere și cifre ale alfabetului ce se începe cu o literă sau cu caracterul de subliniere. În Turbo C se permit următoarele caractere la formarea identificatorilor : Literele mari A..Z, literele mici a..z, cifrele arabe 0..9 precum și caracterul de subliniere '_'. În privința lungimii numelui nu-s nici un fel de limite, dar pentru compilatorul Turbo C au valoare doar primele 32 de caractere. Ex: A, b, x, y, Suma, Gama, Text_no, *beta*, *a1*, *b_1*.

4.2. Descrierea variabilelor.

Toate variabilele din program trebuie să fie descrise. Descrierea variabilelor de obicei se efectuează la începutul programului cu ajutorul instrucțiunilor de descriere.

Instrucțiunea de descriere a variabilelor are următoarea formă generală de înscris: *tip v1,v2,...,vn;*

unde : *tip* - determină tipul valorilor pe care le pot primi variabilele *v1,v2,...,vn*. În calitate de tip se folosește unul din cuvintele cheie deja cunoscute *int*, *float*, *double* sau *char*. Punctul și virgula este semnul sfârșitului instrucțiunii. Între cuvântul cheie ce determină tipul variabilelor și lista numelor variabilelor, trebuie să fie cel puțin un blank. De exemplu:

```
main() {  
    float salary, suma, total;  
    int timc, count;  
    char znak, litera; ... }
```

4.3. Inițializarea variabilelor.

Atribuirea valorii inițiale unei variabile în timpul compilării se numește inițializare. La descrierea variabilelor compilatorului i se poate comunica despre necesitatea inițializării.

Exemple de instrucțiuni de descriere cu inițializarea variabilelor:

```
main() {  
    char bara='\\', litera='T';  
    int anul=2000, luna=9;  
    float alfa,beta,gama=1,7e-12; ...}
```

În ultima instrucțiune de descriere e inițializată doar variabila *gama*, cu toate că la prima vedere pare că toate trei sunt egale cu *1,7e-12*. Din această cauză e mai bine de evitat amestecarea variabilelor inițializate și neinițializate în una și aceeași instrucțiune de descriere.

Există cazuri, când folosirea unei variabile neinițializate poate duce la erori grave din punct de vedere logic la calcularea unor valori. În timpul includerii calculatorului, celulele de memorie nefolosite de sistemul operațional conțin date aleatoare. Când variabila este declarată și ei i se rezervă loc în memorie, conținutul acestor celule de memorie nu se schimbă până când variabila nu este inițializată sau până când ei nu i se atribuie vre-o valoare. Din această cauză, dacă variabila ce n-a fost inițializată și nu i s-a atribuit nici o valoare pe parcursul programului este folosită în calculul unei valori, rezultatul va fi incorect. Acest fapt impune necesitatea inițializării oricărei variabile în timpul declarării acesteia. Aceste inițializări presupun atribuirea valorii 0 (zero) pentru variabile numerice și ' ' (spațiu) pentru cele caracteriale sau de tip șir de caractere.

5. Operații și expresii.

5.1. Operații aritmetice.

În calitate de operații aritmetice limbajul C folosește următoarele:

- | | |
|---------------------|------|
| 1) Adunare | (+). |
| 2) Scăderea | (-). |
| 3) Înmulțirea | (*) |
| 4) Împărțirea | (/). |
| 5) Luarea modulului | (%). |
| 6) Plusul unar | (+). |
| 7) Minusul unar | (-). |

5.2. Operația de atribuire.

Operația de atribuire poate fi simplă sau compusă, în dependență de semnul operației de atribuire. Operația de atribuire simplă se notează cu semnul = și e folosită pentru a atribui variabilei valoarea vreunei expresii. Ex: $x=7$; În limbajul de programare C în expresii se permite folosirea unui număr arbitrar de operații de atribuire. Operațiile de atribuire au o prioritate mai joasă decât cele descrise anterior și se îndeplinesc de la dreapta spre stînga. Operația de atribuire leagă variabila cu expresia, atribuind variabilei valoarea expresiei, ce stă în dreptul semnului de atribuire.

Operațiilor binare: + - * / % >> << & | ^ le corespunde operația de atribuire compusă, care se notează ca $op=$, unde op este una din operațiile enumerate mai sus, de exemplu, += și *= etc.

Operația de atribuire compusă sub forma: `_variabila_op=_expresie_` se definește prin operația de atribuire simplă astfel: `_variabila=_variabila_op_expresie_`

De exemplu, $d+=2$ este echivalent cu $d=d+2$. De regulă pentru operațiile de atribuire compuse calculatorul construiește un program mai efektiv.

5.3. Operații de incrementare(decrementare).

Operația de incrementare notată prin două semne plus consecutive, se aplică asupra unei singure variabile și mărește valoarea cu o unitate. Operația de decrementare se notează prin două semne minus micșorînd valoarea ei cu o unitate. Operația de incrementare poate fi folosită atât ca prefix cât și ca sufix. Diferența dintre acestea constă în momentul schimbării valorii variabilei. Forma prefixă asigură schimbarea variabilei pînă la folosirea ei. Iar forma sufixă - după folosirea ei. Operațiile de incremenatare decrementare se aplică asupra unor variabile de tip ordonat(așa ca *int*). Exemple:

Operația $x++$; asigură incrementarea variabilei x cu o unitate.

În expresia $s=y*x++$; incrementarea lui x se va face numai după calcularea valorii lui s ;

Iar în expresia $s=y*++x$; întâi se va incrementa x și numai după aceea va fi calculată valoarea pentru s .

Și dacă pentru $m=5$ vom scrie expresia $m+++2$, compilatorul tratează primele două plusuri ca forma sufixă de incrementare a variabilei m . Valoarea expresiei va fi 7, și doar apoi variabila m se va mări cu unu. Expresia $(i+j)++$ este greșită căci operația de incrementare se aplică doar către numele variabilei. Operația de incrementare are o prioritate mai înaltă decât cea aritmetică.

5.4. Relații și operații logice.

Operațiile de relație sînt:

- 1) Mai mare ($>$).
- 2) Mai mare sau egal ($>=$).
- 3) Mai mic ($<$).
- 4) Mai mic sau egal ($<=$).
- 5) Egal ($==$).
- 6) Diferit ($!=$).

Operațiile de relație formează o valoare adevărată sau falsă în dependența de faptul în ce raport se află mărimile ce se compară. Dacă expresia de relație e adevărată atunci valoarea ei este unu, în alt caz - 0.

Operațiile de relație au o prioritate mai mică decât cele aritmetice și de incrementare. Între operațiile de relație primele patru au aceeași prioritate și se îndeplinesc de la stînga la dreapta. Ultimele două operații au o prioritate încă mai mică și expresia $a > b == 2$ va primi valoarea unu doar atunci cînd și prima expresia și a doua expresie este adevărată. Operațiile de relație se pot folosi pentru toate tipurile de date de bază, cu excepția șirurilor de caractere (Pentru compararea șirurilor se folosesc funcții incorporate).

Exemple de expresii de relații:

$a > b$;

$(a + b) < 2.5$;

$7.8 <= (c + d)$;

$a > b == 2$;

Expresia de relație $a > b == 2$ întotdeauna va avea valoarea zero (fals). Căci independent de valoarea variabilelor, subexpresia $a > b$ va avea valoarea zero sau unu și compararea oricărei din aceste valori cu 2 ne dă 0.

Astfel observăm că în limbajul de programare Turbo C putem scrie expresii ce par fără sens din punct de vedere al limbajelor de programare tradiționale.

Să cercetăm operațiile logice. Ele, de obicei, se folosesc în calitate de legături pentru reuniune a două sau mai multe expresii. Tabelul ce urmează determină operațiile logice în Turbo C.

Operațiile logice:

Numele operații logice	Semnul operații în Turbo C
1.Conjuncție (SI logic)	&&
2.Disjuncție (SAU logic)	
3.Negare (NU logic)	

Dacă *expr1* și *expr2* sunt careva expresii, atunci

expr1 && *expr2* e adevărată în cazul când ambele sunt adevărate

expr1 || *expr2* e adevărată în cazul când măcar una din expresii e adevărată.

!*expr1* e adevărată dacă *expr1* e falsă și invers.

Noi am stabilit pentru *expr1* și pentru *expr2* cazul cel mai general fără a mai indica ce fel de expresii ele sunt. Deci avem dreptul să scriem: *5&&2*. Valoarea acestei expresii va fi 1, din cauza că în Turbo C valoarea nenulă se tratează ca adevăr iar cea nulă ca fals. Prioritățile operațiilor logice se aranjează în ordine descrescătoare astfel: Negare (!). Conjuncție (&&). Disjuncție (||). Operațiile logice au o prioritate mai mică decât operațiile de relație.

Calcularea expresiilor logice, în care sunt numai operații &&, se termină dacă se descoperă falsitatea aplicării următoarei operații &&. Reșind din definiția semanticii operația &&, cum numai apare valoarea fals prelungirea calculului nu are rost. Analog pentru expresiile logice ce conțin numai operații ||, calculele se termină odată ce apare valoarea adevăr. Exemple de expresii logice:

(5>2)&&47 – adevarata; !(4>7) - adevarata; 4<7 – adevarata;

5.5. Operațiile logice poziționale.

Operațiile logice poziționale se folosesc pentru lucrul cu biți separați sau cu grupuri de biți de date. Operațiile poziționale nu se aplică către datele *float* și *double*. Operațiile logice poziționale:

Numele operației logice poziționale	Semnul operației în Turbo C
<i>ȘI pozițional</i>	&
<i>Sau pozițional</i>	

<i>Sau exclusiv pozițional</i>	\wedge
<i>Deplasare la stînga</i>	\ll
<i>Deplasare la dreapta</i>	\gg
<i>Inversare</i>	\sim

Operația pozițională ȘI se realizează asupra fiecărei perechi de biți a datelor, de exemplu dacă în program avem descrierea variabilelor : *int n,d;* Și variabila *n* are valoarea 5, iar variabila *d* are valoarea 6, atunci expresia *n&d* ne dă valoarea 4, deoarece reprezentarea interioară a valorii variabilei *n* este 0000000000000101 iar a variabilei *d* este 0000000000000110. Rezultatul aplicării operației poziționale ȘI va fi :

n 0000000000000101

d 0000000000000110

4 0000000000000100

Operația pozițională ȘI deseori se folosește pentru evidențierea vreunui grup de biți.

Operația pozițională SAU se aplică asupra fiecărei perechi de biți a datelor și deseori se folosește pentru instalarea unor biți, de exemplu, expresia: $X = X \mid \text{mask}$ instalează în unu acei biți ai lui *x*, cărora le corespunde unu în *mask*. Nu trebuie să confundăm operațiile logice poziționale cu operațiile logice && și || . De exemplu, expresia *1&2* are valoarea 0, iar expresia *1&&2* are valoarea 1.

SAU exclusiv pozițional realizează asupra fiecărei perechi de biți operația de adunare după modulul 2. Rezultatele îndeplinirii operației SAU exclusiv pozițional se determină în tabelul următor :

Bitul primului operand	Bitul operandului al doilea	Bitul rezultatului
0	0	0
0	1	1
1	0	1
1	1	0

Operațiile de deplasare la sînga(>>) și la dreapta (<<) îndeplinesc deplasarea primului operand la stînga sau la dreapta cu un număr de poziții binare determinat de operandul al doilea.

În timpul deplasării la stînga biții, ce se eliberează, se completează cu zerouri. De exemplu, $x \ll 2$ deplasează *x* la stînga cu două poziții, completînd pozițiile eliberate cu zerouri, ceea ce-i echivalent cu înmulțirea la 4. În caz general, a deplasa *x* cu *n* poziții la stînga e echivalent înmulțirii valorii lui *x* cu 2 la puterea *n*.

Deplasarea la dreapta a unei mărimi fără semn (unsigned) aduce la completarea cu zerouri a biților eliberați. Deplasarea la dreapta a unei mărimi cu semn la unele mșini aduce la multiplicarea bitului semnului, iar la altele – biții se completează cu zerouri. Turbo C multiplică bitul semnului și de aceea pentru nemerele pozitive, deplasarea la dreapta cu n poziții e echivalentă împărțirii la 2 la puterea n .

Operația unară inversare (\sim) [tilda] transformă fiecare poziție unară în nulă și invers fiecare poziție nulă în unară.

5.6. Operația dimensiune.

Operația unară dimensiune, notată prin cuvîntul-cheie *sizeof*, dă mărimea operandului său în octeți. Operația dimensiune se folosește sub forma: *sizeof_expresie* sau *sizeof_tip*

Valoarea operației *sizeof* este o constantă întreagă, care determină lungimea tipului rezultatului expresiei. Dacă în calitate de operand al operației *sizeof* folosim *_tip*, atunci primim mărimea obiectului de tipul indicat. În calitate de *_tip* pot fi folosite aceleași tipuri de obiecte, ce se folosesc la descrierea variabilelor.

Operația *sizeof* poate fi folosită peste tot, unde se admite folosirea unei constante întregi. Construcțiile *sizeof_expresie* și *sizeof_tip* se cercetează ca ceva unitar și, astfel, expresia *sizeof_tip -2* înseamnă (*sizeof(_tip)-2*). Sau, de exemplu, expresia *sizeof(a+b+c)+d* e echivalentă expresiei *constana_+d*, unde *constanta_* are valoarea lungimii tipului rezultatului *a+b+c*. Expresia *a+b+c* e luată în paranteze pentru a indica că avem nevoie de lungimea tipului rezultatului *a+b+c*, și nu *a+b+c+d*.

5.7. Operația virgulă.

Operația virgulă (,) permite de a uni cîteva expresii în una singură și, astfel, în Turbo C se introduce noțiunea de expresie cu virgulă, ce are următoarea formă generală : *expresie_expresie_expresie_...* Perechea de expresii, despărțite prin virgulă, se calculează de la stînga spre dreapta. Tipul și valoarea rezultatului expresiei cu virgulă este tipul și valoarea a celei mai din dreapta expresii.

De exemplu: $k=a+b, d=m+n, 5.2+7$ este o expresie cu virgulă și se calculează de la stînga spre dreapta. Valoarea ei este 12.2 de tip *float*. În procesul calculării acestei expresii variabilelor k și d li se atribuie valorile respective. Pentru expresia: $d=(k=5+2, 5+3)$ valoarea variabilei d va fi 8, deoarece ei i se atribuie valoarea expresiei cu virgulă, care, la rîndul său, este egală cu valoarea celei mai din dreapta expresii-operand.

Virgula în Turbo C se folosește în două contexte: ca separator a datelor și ca operație, ce determină calculul consecutiv al expresiilor. De aceea e admisă, de exemplu, așa o expresie: $\text{int } a,b,c=(1,2,5),d;$

unde variabila c se inițializează cu o expresie constantă cu virgulă 1,2,5 și primește valoarea 5. Contextul operației virgulă (separator sau operație) compilatorul “îl simte” după paranteze. În interiorul parantezelor avem operația virgulă, în exteriorul lor – separatorul.

În continuare vom vedea, că o situație analogică poate apărea în lista argumentelor reale în timpul adresării către funcție. Adresarea la funcție, ce conține trei argumente, unde al doilea are valoarea 5, poate arăta, de exemplu, astfel:

$F(a,(t=3,t+2),c);$

5.8. Expresii condiționate.

Expresiile condiționate au următoarea formă generală:

$\text{expr1_}?:\text{expr2_}:\text{expr3_};$

Valoarea expresiei condiționale se calculează astfel: prima se calculează expr1_ . Dacă ea este diferită de zero (e adevărată), atunci se calculează expr2_ și valoarea ei va fi valoarea întregii expresii, în caz contrar se calculează expr3_ . Astfel, semnele de operație $?$ și $:$ determină operația ternară, adică operație cu trei operanzi. De exemplu, pentru a calcula z maximum din a și b e suficient de a scrie expresia:

$z=(a>b)?a:b;$ unde $(a>b)?a:b$ – expresie condițională. Parantezele din jurul primei expresii nu sunt obligatorii, deoarece prioritatea operației ternare $?:$ este foarte joasă, încă mai joasă este prioritatea operației de atribuire. Totuși, se recomandă de a pune parantezele, căci în așa fel condiția se evidențiază vizual. Expresia de mai sus o putem scrie mai puțin efectiv astfel: $a>b?(z=a):(z=b);$

5.9. Conversii de tip.

La scrierea expresiilor ar fi binevoită folosirea datelor omogene, adică a variabilelor și constantelor de același tip. Însă, dacă în expresie se amestecă diferite tipuri de date, atunci compilatorul produce conversia automată a tipurilor în corespundere cu regulile care, în fond, se reduc la următoarele: dacă operația se îndeplinește asupra datelor de diferite tipuri, atunci ambele date se reduc la cel mai superior din aceste două tipuri. O astfel de conversie se numește ridicarea tipului. O consecutivitate de tipuri ordonate de la superior la inferior e determinată în corespundere cu prezența interioară a datelor și arată astfel : double, float, long, int, short, char. Modificatorul unsigned mărește rangul tipului corespunzător cu un semn.

Pentru operația de atribuire (simplă sau compusă) rezultatul calculării expresiei din partea dreaptă se aduce la tipul variabilei căreia i se atribuie această valoare. În acest timp poate avea loc ridicarea tipului sau coborîrea tipului.

Ridicarea tipului de atribuire, de obicei, se produce fără pierderi, pe cînd coborîrea lui poate denatura esențial rezultatul din cauza că acel element de tip superior poate să nu încapă în zona de memorie a elementului de tip inferior.

Pentru păstrarea preciziei calculului la efectuarea operațiilor aritmetice, toate datele de tip float se transformă în double, ceea ce micșorează greșeala de rotundire. Rezultatul final se transformă în tip float, dacă acesta e condiționat de instrucțiunea de descriere respectivă. De exemplu, datele sînt descrise astfel:

float a,b,c; și avem expresia: $a*b+c$

La calcularea valorii expresiei variabilele a , b și c vor fi convertate în double, iar rezultatul va avea tip float. Însă dacă datele sunt descrise astfel : *float a,b; double c;* atunci rezultatul $a*b+c$ va fi de tip double din cauza ridicării tipului.

În afară de conversia automată a tipurilor, îndeplinită de compilator, Turbo C pune la dispoziția programatorului posibilitatea indicării explicite a tipului, la care e necesar de adus o careva mărime sau expresie. Obținem acest lucru folosind operația de conversie a tipului, ce are următoarea formă generală de scriere:

(_tip_)_expresie

Folosirea unei astfel de construcții garantează, că valoarea expresiei va fi convertată la tipul indicat în parantezele din fața expresiei. În această operație în calitate de *_tip_* pot fi folosite aceleași cuvinte cheie, ca și în instrucțiunile de descriere a tipului, împreună cu modificatorii admisibili.

Să cercetăm două expresii în Turbo C.

$d=1.6+1.7$ și $d=(int)1.6+(int)1.7$ cu condiția că variabila d este de tip întreg. În rezultatul îndeplinirii primei expresii valoarea variabilei d va fi 3, deoarece $1.6+1.7$ nu necesită conversia tipurilor și dă rezultatul 3.3 de tip float, iar coborîrea tipului pînă la int, realizată în timpul îndeplinirii operației de atribuire, dă variabilei d valoarea 3 din cauza trunchierii părții fracționare. În rezultaul îndeplinirii expresiei a doua valoarea variabilei d va fi 2, deoarece indică conversia explicită a constantelor *float* în tip *int* pînă la adunarea lor. În timpul îndeplinirii operației de atribuire nu va avea loc conversia tipului, deoarece tipul variabilei coincide cu tipul expresiei.

Operația de reducere a tipului cel mai des se folosește în cazurile, cînd după context nu se presupune conversia automată a tipurilor. De exemplu, pentru asigurarea lucrului corect a funcției *sqrt* ea are nevoie de un argument de tip *double*. De aceea, dacă avem descrierea: *int n;* atunci pentru calcularea rădăcinii patrute din n trebuie să scriem *sqrt((double)n);*

Menționăm, că în timpul aducerii la tipul necesar se convertează valoarea lui n , dar nu se schimbă conținutul ei, căci ea rămâne de tip *int*.

5.10. Prioritățile operațiilor.

Prioritățile operațiilor se cercetau în procesul expunerii lor. În paragraful de față sunt reprezentate operațiile descrise mai sus și e indicată ordinea îndeplinirii lor. Pentru compararea priorităților operațiilor le prezentăm în tabel în ordinea descreșterii priorității. Coloana tabelului “Ordinea execuției” determină consecutivitatea îndeplinirii pentru operațiile cu aceeași prioritate. De exemplu în expresia $k=d+=b-=4$; consecutivitatea îndeplinirii operațiilor se va determina în ordinea de la dreapta la stînga și în rezultat b se va micșora cu 4, d se va mări cu $b-4$, $k=d+b-4$. Ordinea de evaluare, descrisă de o simplă operație de atribuire, va fi următoarea: $b=b-4$; $d=d+(b-4)$; $k=d$;

Prioritatea	Semnul operației	Tipul operației	Ordinea execuției
1	()	Expresie	De la stînga la dreapta
2	! ~ ++ -- - + (unar) sizeof (tip)	Unare	De la dreapta la stînga
3	* / %	Multiplicative	De la stînga la dreapta
4	+ -	Aditive	
5	<< >>	Deplasare	
6	< > <= >=	Relații	
7	== !=	Relații (egalitate)	
8	&	ȘI pozițional	
9	^	SAU exclusiv pozițional	
10		SAU pozițional	
11	&&	ȘI logic	
12		SAU logic	
13	?:	Condiție	
14	= *= /= %= += -= &= >>= <<= ^=	Atribuire simplă și compusă	De la dreapta la stînga
15	,	Virgulă	De la stînga la dreapta

6. Instrucțiuni.

Prin instrucțiune în C se subînțelege o oarecare înscrisoare, ce se termină cu caracterul; (punct și virgulă), sensul căreia determină acțiunile compilatorului în timpul prelucrării textului inițial al programului sau acțiunile procesorului în timpul îndeplinirii programului. Instrucțiunile pot fi împărțite în două grupe:

- 1) Instrucțiunile perioadei de compilare
- 2) Instrucțiunile perioadei de îndeplinire a programului.

La instrucțiuni de compilare se referă instrucțiuni, ce caracterizează datele din program, iar la instrucțiuni de îndeplinire se referă instrucțiuni, ce determină acțiuni de prelucrare a datelor în conformitate cu algoritmul dat. Instrucțiunile de compilare au fost precautate în compartimentele de descriere și declarare a datelor. În continuare vor fi studiate instrucțiunile de îndeplinire.

Fiecare instrucțiune are sintaxa și semantica sa. Sintaxa reflectă metoda înscrierii corecte a construcțiilor instrucțiunii, iar semantica este descrierea acțiunilor efectuate de instrucțiune.

6.1. Tipurile instrucțiunilor.

Instrucțiunea reprezintă o unitate de îndeplinire a programului. Instrucțiuni pot fi simple, compuse și blocuri.

Simplă se numește instrucțiunea, care nu conține în componența sa o altă instrucțiune. La instrucțiunile simple se referă instrucțiunea – expresie, instrucțiunea continuării *continue*, instrucțiunea de terminare *break*, instrucțiunea de întoarcere *return*, instrucțiunea de salt necondiționat *goto*, pe care le vom cerceta mai departe.

Compusă se numește instrucțiunea, ce conține în componența sa alte instrucțiuni. La instrucțiunile compuse se referă instrucțiunea condițională *if – else* instrucțiunile ciclului *for*, *while*, *do while* și instrucțiunea de selecție *switch*.

Bloc – se numește un șir de instrucțiuni luate în acolade({}). Instrucțiunile blocului se îndeplinesc consecutiv în ordinea înscrierii lor în interiorul blocului. La începutul blocului pot fi descrise variabilele interioare. În așa caz se spune, că aceste variabile sunt localizate în interiorul blocului, există și lucrează numai în interiorul blocului și se pierd în afara lui.

În Turbo C instrucțiunea compusă și blocul pot fi folosite oriunde, unde este admisă folosirea instrucțiunii simple. Prin urmare, instrucțiunea compusă poate conține alte instrucțiuni compuse, iar blocul poate conține instrucțiuni compuse și alte blocuri.

Orice instrucțiune poate fi marcată de un identificator, numit *etichetă*. Eticheta este separată de instrucțiune prin două puncte și, astfel, în caz general instrucțiunea are forma: *etichetă_ : corpul_instrucțiunii*;

Eticheta se folosește doar în cazul cînd se folosește saltul necondiționat la aceeașă instrucțiune cu ajutorul instrucțiunii *goto*. În exemplul de mai sus pot lipsi ori eticheta_, ori corpul_instrucțiunii, ori ambele. În cazul lipsei corpul instrucțiunii, avem o instrucțiune vidă, adică așa o instrucțiune ce nu îndeplinește nici o acțiune. În cazul cînd avem lipsa instrucțiunii și prezența etichetei avem o instrucțiune vidă etichetată. Exemplu: *Empty : ;*

Dacă în calitate de corpul instrucțiunii e folosit un bloc, atunci (;) nu se pune. În acest caz, rolul sfîrșitului îl va jucă acolada dreaptă, de închidere (}). De exemplu:

label:{k=a=b;k+=8;}

6.2. Instrucțiuni expresie.

Orice expresie va deveni instrucțiune dacă se va termina cu (;). În așa fel, instrucțiunea-expresie va avea următoarea formă: *expresie_;*

După cum sa menționat, orice instrucțiune poate fi etichetată. Instrucțiunea-expresie aderă la clasa instrucțiunilor simple ai limbajului Turbo C. În rezultatul îndeplinirii în program a instrucțiunii-expresie se calculează valoarea expresiei în conformitate cu operațiile, care sunt definite în ea. De obicei, pot fi una sau mai multe operații de atribuire și atunci instrucțiunea-expresie în limbajul Turbo C are același sens și în alte limbaje. Să studiem secvențe din program unde e folosită instrucțiunea-expresie: Exemplu 1:

*int x,y,z; x=-3+4*5-6; y=3+4%5-6; z=-3*4%-6/5;*

În rezultat vom avea $x=11$, $y=1$, iar $z=0$. Vom scrie ordinea îndeplinirii a acestei secvențe în conformitate cu prioritatea operațiilor, folosind parantezele rotunde:

Pentru $x=-3+4*5-6$: $x=(((-3)+(4*5))-6);$

Pentru $y=3+4\%5-6$: $y=((3+(4\%5))-6)$

Pentru $z=-3*4\%5-6/5$: $z(((((-3)*4)\%(-6))/5));$

Exemplu 2:

int x=2,y,z; x=3+2,x*=y=z=4;*

Aici instrucțiunea-expresie conține o expresie cu virgulă compusă din două subexpresii. În primul rînd se calculează prima subexpresie, apoi- subexpresia a doua. În rezultat vom obține $z=4$, $y=4$, $x=40$. Menționăm faptul, că la declarare x are valoarea 2: $x=(x*(3+2))*(y=(z=4));$

6.3. Instrucțiuni de ramificare (condiționale).

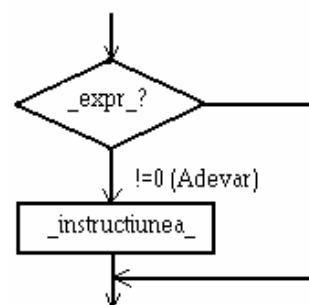
6.3.1. Instrucțiunea de ramificare IF și IF-ELSE.

În structurile ramificate de calcul, unele etape nu întotdeauna se îndeplinesc în una și aceeași ordine. În dependență de careva condiții, care sunt controlate(verificate) pe parcursul calculelor, se aleg pentru executare diferite consecutivități de instrucțiuni. Pentru descrierea astfel de procese în limbajul C se folosesc instrucțiunile ramificate (condiționale). O instrucțiune condițională selectează o singură instrucțiune dintre alternativele sale, pe care apoi o execută. Astfel de instrucțiuni sunt *if* și *if-else*. Instrucțiunea *if* e compusă și sintaxa ei admite unul din următoarele formate:

if(*expresie*)_*instrucțiune* sau
if(*expresie*)_*instrucțiune1*_else_*instrucțiune2*;

În instrucțiunile de ramificare lipsește (;), deoarece construcția *instrucțiune_* care a fost descrisă deja include în sine acest semn (;). Dacă *instrucțiunea_* este simplă, atunci ea se va sfârși cu (;), iar dacă *instrucțiunea_* reprezintă un bloc, atunci acest bloc va fi definit de acolade, unde acolada dreapta va juca rolul sfârșitului. Instrucțiunea *if* lucrează în felul următor:

1. Formatul *if*(*expresie*)_*instrucțiune*. Mai întâi se calculează valoarea expresiei. Dacă rezultatul ei este ADEVĂR (adică expresia != 0), atunci se îndeplinește *_instrucțiunea*, iar în caz contrar se sare peste instrucțiune și nu avem nici o acțiune. Schema algoritmică a formatului *if*(*expresie*)_*instrucțiune* este prezentată pe desen. Fie că *d* este egală cu *c*. Atunci vom mări *d* cu o unitate, iar *c* cu trei unități.



În celelalte cazuri *d* și *c* rămân neschimbate. Instrucțiunea de ramificare a acestui caz:
if(*d==c*) ++*d*,*c*+=3;

În calitate de *instrucțiune_* aici se folosește instrucțiunea expresie cu virgulă. Vom descrie această instrucțiune astfel: *if*(*d==c*)++*d*;*c*+=3;

Diferența constă în aceea că în exemplu al doilea avem 2 instrucțiuni: *if* și instrucțiunea-expresie *c*+=3;

În acest caz dacă (*d==c*) va avea valoare ADEVĂR, atunci totul va rămîne ca mai înainte: *d* se va mări cu o unitate, iar *c* se va mări cu trei unități. În caz contrar, *d* nu se schimbă, și numai *c* se mărește cu 3.

Vom considera acest exemplu din nou:

if(*d==c*){++*d*;*c*+=3;}

În cazul că $(d == c)$ va avea valoarea FALS, d și c vor rămâne neschimbate, deoarece aceste instrucțiuni sînt incluse în acolade, adică formează un bloc, și din punct de vedere logic sunt privite ca o singură instrucțiune bloc.

2. Formatul *if(expresie)_instrucțiune1_else_instrucțiune2*.

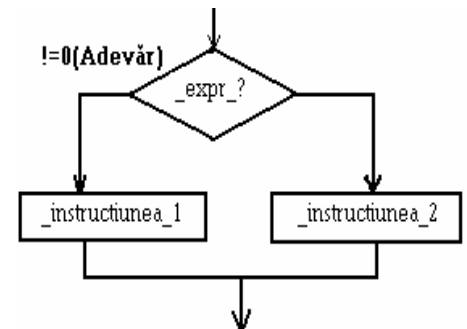
Ca și în cazul precedent, în primul rînd se calculează valoarea expresiei. Dacă ea diferă de zero, adică este ADEVĂR, atunci se îndeplinește instrucțiunea1, în caz contrar se îndeplinește instrucțiunea2.

Schematic această instrucțiune e prezentată pe desen.

De exemplu : fie că e necesar de a calcula z , care este egal cu maximum dintre două numere a și b .

Atunci putem scrie :

if(a>b)z=a;else z=b; Prezența (;) după $z=a$ este necesară, deoarece aceasta este o instrucțiune ce intră în componența instrucțiunii *if*.



Prima formă prescurtată a instrucțiunii *if* ne dă posibilitatea îndeplinirii sau neîndeplinirii oricărei operații, pe cînd a doua formă oferă posibilitatea alegerii și îndeplinirii unei operații din două posibile. Totuși, cîte odată apare necesitatea alegerii unei operații din mai multe. Vom examina un lanț de instrucțiuni:

if(expresie1)_instrucțiune1_else_if(expresie2)_instrucțiune2_else_if(expresie3)_instrucțiune3_else_instrucțiunea4...

Așa un lanț de instrucțiuni e admis, deoarece în locul instrucțiunii în instrucțiunea *if* poate fi orice instrucțiune, inclusiv și *if*. Însă o așa însciere e complicată. Pentru așa procese în C există o instrucțiune specială care va fi studiată ceva mai tîrziu. Să precăutăm 2 exemple:

a) *if(n>0) if(a>b) z=a; else z=b;*

b) *if(n>0){if(a>b) z=a;}else z=b;*

Deosebirea constă în faptul că în cazul a) avem instrucțiunea *if* de formă scurtă, care are ca instrucțiune forma *if_else;*. În cazul b) avem instrucțiunea *if_else* în formă plină, avînd în calitate de *instrucțiune1* forma *if* prescurtată. Secvența b diferă de a numai prin prezența acoladelor, ce definesc un bloc, însă, evident, joacă un rol important la interpretarea acestor instrucțiuni.

6.3.2. Instrucțiunea de salt necondiționat GOTO.

Instrucțiunea *goto* ne dă posibilitatea transmiterii controlului execuției programului la o instrucțiune marcată cu o etichetă. Instrucțiunea *goto* are formatul:

goto_etichetă;

După instrucțiunea *goto*, evident, se îndeplinește instrucțiunea, eticheta căreia coincide cu eticheta din *goto*. Folosirea acestei instrucțiuni în Turbo C nicidecum nu se deosebește de folosirea ei în alte limbaje algoritmice. Însă e dorită folosirea cât mai redusă, a acestei instrucțiuni, deoarece limbajul Turbo C face parte din clasa limbajelor structurate. Totodată, avînd la dispoziție instrucțiunile *goto* și *if*, programatorul poate programa diferite operații complicate.

Exemplu :Folosirea instrucțiunilor *if* și *goto* la organizarea ciclurilor. Acest program calculează valoarea lui *y* care este egală cu $\sum_{n=1}^{50} n/(n+5)$, unde $n=1, \dots, 50$

```
#define lim 50
main() {
  int n=0; float y=0;
  m1:++n;
  if(n<=lim) {
    y+=n/(n+50); goto m1;} }
```

Prima linie a programului e menită preprocesorului, căruia îi se indică valoarea constantei *lim*=50. Astfel se procedează în cazul cînd vom necesita schimbarea limitei de sumare. Pentru aceasta trebuie schimbată doar directiva *#define*, iar substituirile corespunzătoare în textul programului preprocesorul le va face fără participarea noastră.

Programul constă dintr-o singură funcție *main()*, corpul căreia e definit cu ajutorul acoladelor exterioare. Instrucțiunea *if* are aici forma prescurtată, avînd ca instrucțiune un bloc, care conține instrucțiunea *goto*, ce ne dă posibilitatea calculării ciclice a sumei.

În încheiere menționăm că utilizarea *if* și *goto* la organizarea ciclurilor e un semn de cultura insuficientă de programare. Exemplu dat arată numai cum se poate organiza un ciclu cu ajutorul instrucțiunilor *if* și *goto*, dar în nici un caz nu servește drept exemplu pentru copiere. La organizarea ciclurilor în limbajul de programare Turbo C se folosesc alte posibilități, cu mult mai fine.

6.3.3. Instrucțiunea de selectare SWITCH.

Instrucțiunea de selectare e destinată pentru selectarea unei singure variante dintr-o mulțime posibilă de ramificări în program. Orice ramificare poate fi organizată cu ajutorul lanțului de instrucțiuni *if_else_if_..._else_*, însă în cazurile cînd sunt prezente mai mult de 3 condiții e mai rațională folosirea lui *switch*. Instrucțiunea de selectare este compusă și are formatul prezentat mai jos:

```
switch (expresie) {
  case expr_const1:instrucțiune1;
  case expr_const2:instrucțiune2; ...
```

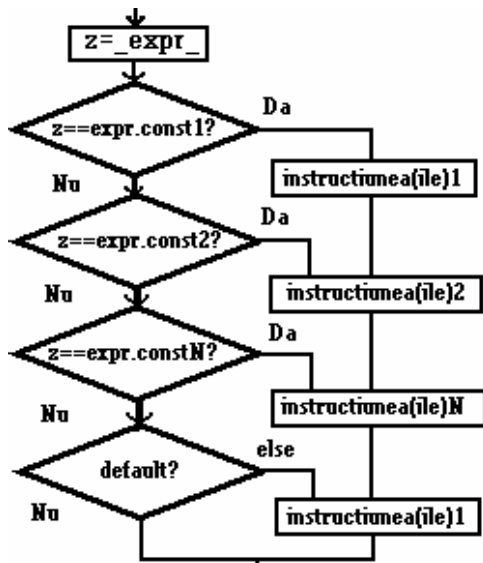
```

case expr_const_n:instrucțiune_n;
default:instrucțiune; }

```

După cuvîntul cheie *switch* în paranteze este scrisă o expresie, valoarea căreia trebuie să fie de tip *int* sau *char*. Mai departe în acolade se înscriu instrucțiunile – variantele marcate cu prefixele: *case expresia_constantă*: unde expresia constantă deasemenea trebuie să fie de tip întreg, iar toate prefixele trebuie să fie diferite. O variantă poate avea prefixul *default*, prezența careia nu este obligatorie, dar în majoritatea cazurilor acest prefix este prezent în construcția *switch*.

Instrucțiunea de selectare lucrează în felul următor. La început se află valoarea expresiei din paranteze, apoi această valoare se compară cu expresiile constante ale case prefixelor și se îndeplinește varianta i, pentru care aceste valori coincid. După



ce sau îndeplinit instrucțiunile ce se conțin în case-ul respectiv, se îndeplinesc instrucțiunile variantei următoare și așa pînă ce se termină toate instrucțiunile variantei. Dacă valoarea expresiei în *switch* nu coincide cu nici o valoare a case prefixelor, atunci ori se îndeplinește instrucțiunea variantei cu prefixul *default* (dacă *default* este prezent), ori nu se îndeplinește nici o variantă (dacă *default* este absent). Schema algoritmică de îndeplinire a instrucțiunii *switch* este prezentă pe desen.

Să examinăm exemplul de întrebuințare a instrucțiunii *switch* : se scrie un fragment de program, ce tipărește patru rînduri din poezie, începînd de la rîndul *k*:

```

switch(k) {
case 1: printf("A fost o dată ca-n povești,\n");
case 2: printf("A fost ca niciodată,\n");
case 3: printf("Din rude mari împărățești,\n");
case 4: printf("O preafrumoasă fată.\n");
default printf("Poezia nu conține rînd cu așa număr"); }

```

În acest exemplu e folosită funcția *printf()*, ce asigură tiparul rîndului cu format. Dacă *k* ar fi egal cu 3, atunci vom avea ca rezultat :

```

"Din rude mari împărățești
O preafrumoasă fată.
Poezia nu conține rînd cu așa număr "

```

6.3.4. Instrucțiunea de întrerupere BREAK.

În practica programării câteodată apare necesitatea îndeplinirii numai a unei variante case fără îndeplinirea celor ce urmează după ea, adică trebuie întreruptă logica stabilită de lucrul a instrucțiunii switch. Evident că pentru așa ceva e necesar introducerea unei instrucțiuni între instrucțiunile variantei, care ar servi ca sfârșit al îndeplinirii lui switch. Așa o instrucțiune este BREAK, executarea căreia rovoacă terminarea instrucțiunii switch. În acest caz instrucțiunea switch va avea următorul format:

```
switch (expresie) {  
  case expr_const1:instrucțiune1; break;  
  case expr_const2:instrucțiune2; break ;  
  .....  
  case expr_const_n:instrucțiune_n; break;  
  default:instrucțiune; break; }
```

Instrucțiunea *BREAK* se înscrie în variante atunci când este nevoie de ea. Îndeplinirea ei aduce la trecerea controlului la instrucțiunea următoare după switch. La prima vedere pare că nu este necesar de a scrie instrucțiunea *BREAK* după varianta default. Default poate fi situat în orice loc al instrucțiunii switch, chiar și pe primul loc între variante, și atunci *BREAK* este necesar pentru a evita îndeplinirea celorlalte variante. Exemplu: Schimbăm formularea problemei precedente în felul următor : fie că e necesar de tipărit rîndul *k* dinte cele patru rînduri ale poeziei date:

```
switch(k){  
  case 1 : printf("A fost o dată ca-n povești,\n");break;  
  case 2 : printf("A fost ca niciodată,\n");break;  
  case 3 : printf("Din rude mari împărătești,\n");break;  
  case 4 : printf("O preafrumoasă fată.\n");break;  
  default:printf("Poezia nu conține rînd cu așa număr\n"); }
```

Trebuie de menționat, că instrucțiunile din variantele *case* sau *default* pot lipsi. Aceasta este necesar când avem nevoie de a obține același rezultat la trecerea la diferite prefixe. Exemplu:

```
switch (L) {  
  case `C`:  
  case `c`: printf("Calculator\n");break; }
```

În acest caz, dacă $L='c'$ va fi tipărit cuvîntul "calculator". În cazul când $L='C'$, atunci va fi îndeplinită instrucțiunea ce urmează după primul case, dar deoarece aici lipsește instrucțiunea, se vor îndeplini instrucțiunile situate mai jos, pînă ce nu se întîlnește instrucțiunea break.

6.4. Instrucțiuni iterative(ciclice).

Instrucțiunile precăutate mai sus redau operații care trebuie efectuate conform algoritmului și fiecare din ele se îndeplinesc numai odată. În cazul, când una și aceeași instrucțiune trebuie să fie executată de n ori cu diferite valori ale parametrilor se folosesc instrucțiunile ciclice. Distingem 3 instrucțiuni ciclice în C :

- 1) Instrucțiunea ciclică cu parametru (FOR)
- 2) Instrucțiunea ciclică precedată de condiție (WHILE)
- 3) Instrucțiunea ciclică cu postcondiție (DO-WHILE)

6.4.1. Instrucțiunea ciclică FOR.

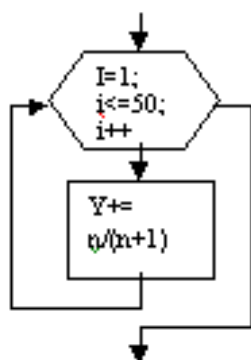
Ciclul FOR posedă următoarele caracteristici:

numărul de repetări ale ciclului este cunoscut de la începutul executării lui;
conducerea ciclului este efectuată cu ajutorul unei variabile de tip int, numită parametrul ciclului, care, în acest proces ciclic primește valori consecutive de la valoarea inițială dată până la valoarea finală dată. Sintaxa instrucțiunii este următoarea: *for(expresie1;expresie2;expresie3) instrucțiune;* unde expresie1– expresie de inițializare a parametrului ciclului, expresie2- expresie de control, expresie3- expresie de incrementare/decrementare(corecție) a parametrului ciclului.

Instrucțiunea ciclică *for* lucrează în felul următor: la început se calculează expresia de inițializare. Apoi, dacă expresia de control este adevărată, atunci se îndeplinește instrucțiunea. După îndeplinirea instrucțiunii se execută expresia de corecție și din nou se controlează expresia de control, justetea căreia duce la îndeplinirea repetată a instrucțiunii. Dacă expresia de control are valoarea falsă atunci îndeplinirea ciclului *for* se termină, adică controlul se transmite instrucțiunii programului ce urmează după instrucțiunea *for*. De exemplu: de calculat $y = \sum i/(i+1)$; unde $i=1..50$;

$y=0; \text{for}(i=1; i \leq 50; i++) \{ y += i/(i+1) \};$

aici $i=1$ este expresie de inițializare, $i \leq 50$ – expresie de control, $i++$ - expresie de corecție. Acoladele mărginesc corpul ciclului (instrucțiunea). În caz când corpul ciclului e compus numai dintr-o instrucțiune, acoladele nu sunt necesare.



Schema algoritmică îndeplinirii instrucțiunii *for* este prezentată pe desen. Din schema algoritmică se vede, că instrucțiunea *for* este un ciclu cu precondiție : decizia de a îndeplini din nou ciclul sau nu se ia înaintea începerii îndeplinirii lui și evident, se poate întâmpla ca corpul ciclului să nu fie îndeplinit nici o dată.

Cîteodată apare necesitatea ieşirii din ciclu înainte de termen. Pentru aceasta în corpul ciclului, în locul unde se doreşte ieşirea din ciclu se foloseşte instrucţiunea *BREAK*, după îndeplinirea căreia are loc transmiterea controlului la instrucţiunea următoare după ciclu.

Limbajul C nu pune restricţii la tipul instrucţiunilor din cadrul corpului ciclului. În aşa mod, corpul ciclului poate fi alcătuit din instrucţiuni şi simple şi compuse, în particular- corpul unui ciclu poate fi alt ciclu. În unele algoritme apar situaţii, cînd e necesară imbricarea unui ciclu în altul. De exemplu la prelucrarea unei matrice un ciclu exterior răspunde de prelucrarea rîndurilor, iar altul interior – a coloanelor. În acest caz sintaxa va fi următoarea:

```
For(i=1;i<=n;i++){
  For(j=1;j<=m;j++){
    corpul ciclului } };
```

unde n-numărul de rînduri în matrice, m- numărul de coloane, acoladele interioare mărginesc corpul ciclului cu parametrul j, iar acoladele exterioare mărginesc corpul ciclului exterior cu parametru i.

6.4.2. Instrucţiunea ciclică WHILE.

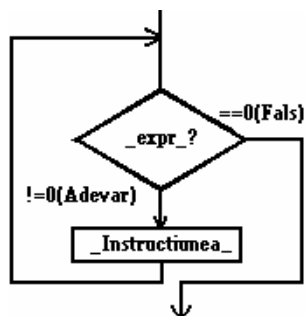
Ciclul *while* este folosit în cazul, cînd nu este cunoscut numărul de repetări ale ciclului şi nu există necesitatea ca ciclul să fie executat măcar o singură dată.

Instrucţiunea de ciclare *while* are următoarul format:

```
while (expresie) instrucţiune;
```

Instrucţiunea de ciclu *while* lucrează în felul următor: dacă expresia este adevărată (sau diferită de zero, ce din punct de vedere al limbajului Turbo C este una şi aceeaşi), atunci instrucţiunea se îndeplineşte o dată şi apoi expresia din nou se testează.

Această succesiune de acţiuni, ce constă în testarea expresiei şi îndeplinirea instrucţiunii, se repetă periodic pînă ce expresia nu devine falsă(din punct de vedere al limbajului Turbo C devine egală cu zero). Instrucţiunea se numeşte corpul ciclului şi în majoritatea cazurilor reprezintă un bloc, în componenţa căruia intră cîteva



instrucţiuni. Schema algoritmică a executării instrucţiunii *while* e prezentată pe desen.

Se observă, că instrucţiunea *while* este un ciclu cu precondiţie. Testul de control este executat înainte intrării în corpul instrucţiunii. De aceea este posibil ca corpul ciclului nu va fi îndeplinit niciodată. În afară de aceasta, pentru a nu admite îndeplinirea la infinit a ciclului, ci numai de anumite

ori, este necesar la fiecare îndeplinire nouă a ciclului de a modifica variabila parametru , ce intră în componența expresiei. Spre deosebire de ciclul for, unde variabila parametru putea fi numai de tip întreg, parametrul ciclului while poate fi și de tip float, adică pasul ciclului poate fi diferit de 1 și chiar un număr fracționar. Exemplu:

```
i=1; while(i<=50) {y+=i(i+1); i++; } ;
```

Ca și în cazul ciclului for, corpul unui ciclu while poate fi de asemenea un ciclu. Exemplu:

```
i=1; while(i<=n) { j=1; while(j<=m) { corpul ciclului; j++; } i++; }
```

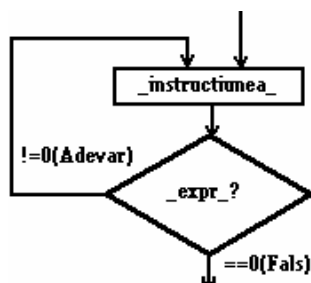
6.4.3. Instrucțiunea ciclică DO_WHILE.

Instrucțiunea ciclică *DO_WHILE* se folosește în cazul când numărul de repetări ale ciclului nu-i cunoscut, dar în același timp e necesar ca ciclul să fie îndeplinit măcar o singură dată. Instrucțiunea de ciclare *do_while* are următoarea formă:

```
do instrucțiune while(expresie);
```

Instrucțiunea *do_while* lucrează în felul următor: la început se îndeplinește instrucțiunea, apoi se calculează valoarea expresiei. Dacă valoarea expresiei este adevărată, atunci instrucțiunea se îndeplinește din nou, dacă expresia este falsă, atunci îndeplinirea ciclului se termină.

Schema algoritmică de îndeplinire a instrucțiunii *do_while* este reprezentată pe desen. Instrucțiunea *do while* determină un ciclu cu postcondiție, deoarece controlul necesității îndeplinirii repetate a instrucțiunii are loc după îndeplinirea corpului ciclului și, astfel, corpul ciclului întotdeauna se îndeplinește cel puțin o dată. Analogic, ca și pentru ciclul *while*, programatorul trebuie să aibă grijă de terminarea ciclului, schimbând parametrul ciclului în corpul ciclului. Parametrul ciclului *do while*, ca și parametrul ciclului *while* poate fi de tip float, fapt ce permite de a opera cu pași zecimali la organizarea ciclului.



Exemplu: $i=1; do \{y+=i(i+1); i++;\} while(i \leq 50);$

Ciclul *do_while* de asemenea poate fi imbricat:

```
i=1; do { j=1;
do {corpul ciclului; j++;} while(j<=m)
i++; }
while(i<=n);
```

Deci, fiind cunoscute cele 3 instrucțiuni ciclice FOR, WHILE și DO_WHILE le putem folosi în diferite scopuri aparte sau împreună pentru cazuri concrete. În cazul când cunoaștem numărul de repetări al ciclului, pasul indexului fiind =1 folosim instrucțiunea ciclică FOR. În cazul când numărul de repetări al ciclului e necunoscut,

dar corpul ciclului trebuie să fie executat măcar o singură dată folosim instrucțiunea ciclică *DO_WHILE*. Și în cazul când nu este cunoscut numărul de repetări al ciclului, iar corpul ciclului e necesar să fie executat de 0 sau mai multe ori, în dependență de o condiție folosim instrucțiunea ciclică *WHILE*. Mai mult ca atât, în cazul ciclurilor imbricate sunt posibile combinații de instrucțiuni ciclice: *for-while*; *do_while-while*; s.a.

6.4.4. Instrucțiunea de continuare *CONTINUE*.

Instrucțiunea *CONTINUE* este folosită în corpul ciclului cu scopul de a preda controlul la începutul ciclului. Există cazuri, când la îndeplinirea a careva condiții trebuie de întrerupt executarea iterației curente și de trecut la îndeplinirea iterației următoare a ciclului. În așa cazuri este folosită instrucțiunea *CONTINUE*. Instrucțiunea de continuare are următoarea formă: *continue*;

Instrucțiunea *continue* poate fi realizată în toate cele trei tipuri de cicluri, dar nu și în instrucțiunea *switch*. Ea servește la depășirea părții rămase a iterației curente a ciclului, ce o conține nemijlocit. Dacă condițiile ciclului admit o nouă iterație, ea se îndeplinește, în caz contrar el se termină. Să precutăm următorul exemplu:

```
int a,b; b=0;
for(a=1;a<100;a++) {b+=a; if (b%2) continue;
... prelucrarea sumelor pare ... }
```

În cazul, când suma *b* va fi impară, operatorul *continue* transmite controlul iterației următoare a ciclului *for* fără a executa partea următoare a corpului ciclului, unde are loc prelucrarea sumelor pare.

7. Masive.

7.1. Descrierea masivelor.

Masivul reprezintă un șir ordonat de elemente de același tip. Faptul, ca masivul este un tot întreg, compus din câteva elemente ne permite să numim variabila de tip masiv variabila de tip compus.

Masivul se poate caracteriza prin nume, tip, dimensiune. Formatul comun de descriere a masivelor este:

tip nume[d1][d1]...[dn]; unde :

tip este tipul comun pentru toate elementele masivului, adică tipul masivului. Tip al unui masiv poate fi orice tip de date deja definit: întreg, real, caracterial ș.a.

nume este numele masivului. În calitate de nume al masivului este folosit orice identificator. Mai mult ca atât, deoarece numele masivului este identificator, asupra lui se răspîndește totul ce-i indicat în compartimentul "Nume de variabile

(identificatori)”, $d1, d2, dn$ - dimensiunile masivului. Dimensiunea masivului indica numarul de elemente prezente in masiv. Dimensiunea masivului poate fi o expresie constanta cu rezultat intreg. In dependenta de cantitatea de dimensiuni, masivele se clasifica in:

1. masive unidimensionale(cu 1 dimensiune);

masivul unidimensional reprezintă un șir de elemente aranjate uniform într-un rând. Fiecare element al unui masiv unidimensional are 1 coordonată: numărul de ordine a elementului în șir.

2. masive bidimensionale (cu 2 dimensiuni);

masivul bidimensional reprezintă o structură formată din rânduri și coloane. Fiecare element al unui masiv bidimensional are 2 coordonate: numărul rândului și numărul coloanei.

3. masive tridimensionale (cu 3 dimensiuni);

masivul tridimensional reprezintă o structură echivalentă cu un cub în volum cu 3 dimensiuni: lungimea, lățimea, înălțimea. Fiecare element al unui masiv tridimensional are 3 coordonate: numărul rândului(în lungime), numărul coloanei(în lățime) și numărul înălțimei(în adâncime).

4. masive multidimensionale.

Exemple de descriere a masivelor:

int vector[20]; *vector - masiv unidimensional din 20 de numere întregi;*
float x[10]; *x – masiv cu 10 elemente de tip întreg;*
float matrix[7][9]; *matrix - masiv bidimensional din 63 (7*9) de numere flotante;*
char fraza[25]; *fraza - masiv(propoziție) din 25 de caractere ;*
int spase[15][30][18]; *space - masiv tridimensional de numere întregi (masiv unidimensional de masive bidimensionale);*

7.2. Accesul la elementele masivului.

Cu toate că masivul este un tot întreg, nu se poate vorbi despre valoarea masivului întreg. Masivele conțin elemente cu valorile cărora se operează în program. Fiecare element în masiv își are indicele și valoarea sa. În calitate de indice a unui element se folosește un număr întreg ce indică numărul de ordine al elementului în masiv. Enumerarea elementelor în masiv conform numărului de ordine se începe de la zero. Deci, indicele unui element poate avea valori de la 0 pâna la $d-1$, unde d este dimensiunea masivului.

În calitate de valoare a unui element din masiv poate servi orice număr de tipul indicat la descrierea masivului, adica tipul valori atribuită oricărui element din masiv trebuie să fie compatibil cu tipul masivului. Sintaxa de acces la orice element a unui

masiv este următoarea: *nume[i1][i2]..[in]*. Unde *nume* este numele masivului, *i1* – indicele elementului în dimensiunea 1, *i2* – indicele elementului în dimensiunea 2, *in* – indicele elementului în dimensiunea *n*. În cele mai dese cazuri se operează cu masive unidimensionale și bidimensionale. Accesul la un element al unui masiv unidimensional se face în felul următor: *nume[i]*; unde *nume* - numele masivului, *i* - numărul de ordine a elementului în masiv. Exemplu:

vector[5]; se accesează elementul cu numărul de ordine 5 din masivul vector.

fraza[20]; se accesează elementul cu indicele 20 din masivul fraza.

Accesul la un element al unui masiv bidimensional se face prin *nume[i][j]*; unde *i* este numărul rîndului pe care se află elementul; *j* este numărul coloanei pe care se află elementul. Exemplu:

matrix[4][7]; se accesează elementul de pe rîndul 4 și coloana 7 a masivului matrix.

y[0][0]; se accesează primul element al masivului, adică rîndul 0, coloana 0;

În cazul cînd masivul este de tip simplu, atribuirea valorii unui element al masivului se face ca și în cazul atribuirii valorii unei variabile simple. Exemple:

x[0]=7.125;

vector[19]+=1;

matrix[1][1]=5.5;

fraza[3]='b';

space [3][5][2]=8;

În cazurile cînd masivul este de tip structurat, atribuirea valorii și accesul la un element al masivului se face conform regulilor de atribuire și accesare pentru variabile structurate.

Un element al masivului poate apărea în orice expresie, unde e admisibilă prezența unei variabile de tip compatibil cu tipul valorii elementului.

7.3. Inițializarea masivelor.

Deseori e necesar ca elementele masivului să posede valori chiar la momentul descrierii masivului.

Procesul de atribuire a valorilor elementelor masivului în timpul descrierii lui se numește inițializarea masivului. Sintaxa de inițializare a unui masiv unidimensional este:

tip nume[d]={v0,v1,v2,...,vn-1}; unde *tip* este tipul masivului, *nume* este numele masivului, *v0,v1,v2,vn-1* valorile respective ale elementelor *nume[0],nume[1]* etc. Exemplu:

int x[8]={1,3,15,7,19,11,13,5};

În acest caz elementele masivului vor avea următoarele valori: $x[0]=1$; $x[1]=3$; $x[2]=15$; $x[3]=7$; $x[4]=19$; $x[5]=11$; $x[6]=13$; $x[7]=5$;

E de menționat faptul, că indicii masivului se schimbă începând dela zero. Adică la descrierea masivului valoarea maximă a indicelui masivului coincide cu numărul de elemente în masiv minus unu.

La inițializarea masivului nu e numai decît de indicat dimensiunile masivului. Compilatorul va determina numărul elementelor după descrierea masivului și va forma un masiv cu mărimea respectivă. De exemplu:

```
int x[]={1,3,15,7,19,11,13,5};
```

Elementele masivului vor primi valori ca și în cazul precedent.

Vom examina încă cîteva exemple de inițializare a masivelor:

```
float vector[4]={1.2,34.57,81.9,100.77}; // vector – masiv din 4 elemente de tip float;
```

```
int digit[5]={1,2,3}; // digit – masiv de tip întreg din 5 numere, ultimelor două elemente li se atribuie valoarea zero.
```

```
char m[5]={'A','B','C','D'}; // m – masiv din 5 caractere, ultimul element are valoarea nul-caracter;
```

```
float const y[4]={25,26,17,18}; // inițializarea masivului y[4], elementele căruia sunt constante de tip float și nu pot fi schimbate în decursul îndeplinirii programului.
```

Să examinăm inițializarea masivului bidimensional:

```
Int a[3][3]={ {1,4,2},  
              {7,5,3},  
              {8,6,9} };
```

Inițializarea masivului bidimensional se efectuează pe rînduri. Elementele acestui masiv au următoarele valori: $a[0][0]=1$; $a[0][1]=4$; $a[0][2]=2$; $a[1][0]=7$; $a[1][1]=5$; $a[1][2]=3$; $a[2][0]=8$; $a[2][1]=6$; $a[2][2]=9$;

La inițializarea masivului dat fiecare rînd se include în acolade . Dacă în rîndurile indicate de noi nu vor ajunge elemente pentru completarea rîndurilor, în acest caz în locul elementelor pentru care n-au ajuns valori vor apărea zerouri.

Dacă în acest exemplu vom omite acoladele interioare, rezultatul va fi același. Dacă lipsesc acoladele interioare ,elementelor masivului se vor atribui valorile în mod succesiv extrase din listă .Completarea masivului se efectuiază pe rînduri. Elementele masivului pentru care în listă n-au ajuns valori, primesc valorile zero. Dacă în listă sînt mai multe valori decît elemente, atunci o astfel de listă se socoate greșită. Cele menționate mai sus se referă la toate tipurile de masive.Exemple:

Inițializarea masivului bidimensional:

```
int a[3][3]={ 1,4,2,7,5,3,8,6,9};
```

Trei metode echivalente de inițializare a masivului tridimensional:

```
int p[3][2][2]={ { {1,2},{3,4} }, { {5,6},{7,8} }, { {9,10},{11,12} } };
```

```
int p[3][2][2]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

```
int p[3][2][2]={1,2,3,4,5,6,7,8,9,10,11,12};
```

7.4. Exemple de prelucrare a masivelor.

Prezentăm 2 exemple de prelucrare a masivelor unidimensional și bidimensional:

Exemplu 1. Prelucrarea unui masiv unidimensional:

Este dat un masiv unidimensional x cu n elemente. Comparați suma primei jumătăți a masivului cu media aritmetică jumătății a doua a masivului.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
void main(void) {
    int x[20],n,k,i,s=0;
    float m,s1=0,r=0;
    printf("\nCulege mărimea masivului n<=20\n");
    scanf("%d",&n);
    for(i=0;i<n;i++){
        printf("Culege elementul %d\n",i);
        scanf("%d",&x[i]);}
    printf("Masivul inițial este:\n");
    for(i=0;i<n;i++){
        printf("%d ",x[i]);}
    if (fmod(n,2)==0) k=floor(n/2); else k=floor(n/2)+1;
    for(i=0;i<n;i++){
        if (i<k) s+=x[i]; else {s1+=x[i]; r++;} }
    m=s1/r;
    printf("\nSuma primei jumătăți este %d\n",s);
    printf("Media la a doua jumătate este %f",m);
    getch();}
```

Exemplu 2. Prelucrarea unui masiv bidimensional:

Este dat un masiv bidimensional. $X[n,n]$.

Calculați produsul elementelor pozitive pare din aria hașurată.



```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
void main(void) {
    int x[20][20],n,i,j,p,k;
```



```

printf("\nCulege mărimea masivului n<=20\n");
scanf("%d",&n);
printf("\nCulege elementele masivului\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("\nCulege elementul x[%d][%d]\n",i,j);
scanf("%d",&x[i][j]);}}
printf("\nMasivul inițial este:\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%d ",x[i][j]);}
printf("\n");}
p=1; k=floor((float)n/float(2));
for(i=0;i<n;i++){
for(j=k;j<n;j++){
if ( (x[i][j]>0)&&(fmod(x[i][j],2)==0) ) p=p*x[i][j];
}} printf("\nProdusul elementelor din aria hașurata este=%d k=%d\n",p,k);
getch(); }

```

8. Șiruri de caractere.

Numim șir o succesiune de caractere ale alfabetului, adică o propoziție. Spre deosebire de alte limbaje de programare, limbajul C nu conține un tip de date special ce desemnează șiruri de caractere. Limbajul C operează cu șirurile cum ar lucra cu o succesiune de date de tip caracter amplasate într-un masiv. Aici fiecare simbol din șir este o componentă aparte a masivului. Deci, pentru a defini o variabilă, valoarea careia va fi un șir de caractere, în limbajul C trebuie de declarat un masiv de tip char cu o lungime egală cu cantitatea maximal posibilă de caractere în șir. Următorul exemplu arată cum s-ar putea declara un asemenea masiv cu scopul de a culege un nume de la tastatură și apoi de-l afișat la monitor:

```

void main(void) {
char a[20]; int i;
printf("Culege numele");
for(i=0; i<20; i++)
scanf("%c",a[i]);
printf("Numele dumneavoastra este:");
for(i=0; i<20; i++)
printf("%c",a[i]); }

```

În acest program se observă multe neajunsuri și incomodități. Pentru a culege numele aici este folosit un ciclu cu 20 repetări care citește câte un caracter de la tastatură și-l înscrie în celula corespunzătoare a masivului a[20]. Rezultă că cu ajutorul acestui program se poate culege numai nume alcătuite din 20 de caractere,

sau nume din k caractere cu $20-k$ spații după ele. Iar afișarea numelui este făcută cu ajutorul funcției *printf()* inclusă în ciclu, care afișază la monitor câte un caracter din numele cules.

Notă: *La declararea masivului de tip char pentru descrierea unui șir se indică mărimea masivului cu o celulă mai mult de cât lungimea maximă presupusă a șirului din cauză că ultima celulă din masiv este rezervată pentru simbolul nul "/0".*

Evident că imposibil de programat într-un limbaj de nivel înalt folosind astfel de mecanisme. Din cauza că programele pentru prelucrarea informației textuale au o popularitate foarte mare, limbajul C conține mecanisme care ușurează lucru cu șirurile de caractere. Aici șirurile sînt precăutate ca un tip special de masive, fapt ce permite introducerea și afișarea șirurilor ca un tot întreg. Pentru introducerea șirului în memoria calculatorului este folosită funcția *gets()*. Această funcție are următoarea sintaxă:

gets (nume); unde *nume* este parametrul funcției și reprezintă numele variabilei tip șir, adică a masivului de tip char. Exemplu:

```
void main(void) {  
    int i; char name [15];  
    printf ("Culege numele:");  
    gets(name);  
    printf ("Numele dumnevoastra este:");  
    for (i=0; i<15; i++)  
        printf ("%C", name[i]); }
```

Aici funcția *gets()* va precăuta primele 14 caractere culese de la tastatură ca valoare a șirului cu nume *name*, iar ultima celulă din masiv va conține simbolul nul “\0”. În timpul lucrului funcției *gets()* executarea programului se stopează. Funcția *gets()* așteaptă pînă utilizatorul va culege un șir de la tastatură. Pentru ca textul cules să fie atribuit ca valoare unei variabile de tip șir, utilizatorul trebuie să culeagă tasta ENTER. După aceasta propoziția culeasă va deveni valoare a variabilei de tip șir, iar cursorul va trece în rînd nou pe monitor. Așadar în timpul culegerii tastei ENTER compilatorul C adaugă la sfîrșitul șirului simbolul nul. În exemplu de mai sus culegerea șirului se precăută ca culegerea unei variabile de tip șir aparte și nu ca culegerea a mai multor variabile de tip caracter. Însă afișarea numelui rămîne incomodă. Aici, dacă vor fi culese mai puțin de 15 simboluri de la tastatură, atunci elementele masivului *name[15]* ce urmează după simbolul nul vor conține mărimi aleatoare.

În afară de funcția *gets()* limbajul C mai conține o modalitate de a introduce o propoziție în memorie ca valoare a unei variabile de tip șir. Folosind funcția de

introducere cu format *scanf()*, se poate aplica următoarea sintaxă: *scanf("%s",name);* care așteaptă culegerea de la tastatură a unui șir de caractere, pe care apoi (după culegerea tastei ENTER îl atribuie ca valoare variabilei *name*. Aici *%s* este formatul tipului șir de caractere.

Pentru afișarea la monitor a unui șir de caractere este folosită funcția *puts()*. Funcția *puts()* poate avea în calitate de parametru numai un șir de caractere. Sintaxa funcției este următoarea: *puts(parametru);* unde în calitate de valoare poate fi folosit un șir de caractere sau numele unei variabile de tip șir de caractere. Exemplu:

```
puts("Elena"); puts(name);
```

Majoritatea compilatoarelor C trec cursorul din rând nou după executarea funcției *puts()*. Însă există și așa versiuni de compilatoare, care nu îndeplinesc această trecere din rând nou. În acest caz e binevoită folosirea simbolului de trecere în rând nou "*\n*". Exemplu: *puts("Elena\n");*

Folosind funcțiile *gets()* și *puts()* exemplul de mai sus se poate scrie în felul următor:

```
void main (void) {  
char name[15];  
puts (" Culege numele");  
gets (name);  
puts ("Numele dvs este:");  
puts (name); }
```

8.1. Masive de șiruri.

Declarînd un masiv *char S[20]* putem păstra în el valoarea unui șir de caractere. În cazul, cînd este nevoie de prelucrat o totalitate din cîteva șiruri, e comod de folosit masive de șiruri.

Un masiv de șiruri este un masiv bidimensional tip *char* compus din linii și coloane. Pe fiecare linie din așa masiv va fi înscris cîte un șir de caractere. Numărul maximal de șiruri astfel înscrise în masiv va fi egal cu cantitatea liniilor din masiv.

De exemplu: *char propoziție[10][35]* este un masiv, în care pot fi scrise 10 variabile de tip șir de caractere, fiecare avînd lungimea maximală de 34 de caractere. Pentru a accesa un șir din așa masiv se va folosi sintaxa: *propoziție[i]*, unde *i* este numărul rîndului din masiv unde se va afla șirul. Pentru a accesa un caracter al șirului *i* din masiv se va folosi sintaxa: *propoziție[i][j]* unde *j* este poziția caracterului în șirul *i*.

9. Structuri în C/C++.

Pînă în momentul de față au fost studiate tipurile de date compuse, elementele cărora aparțin aceluiași tip de date (simplu sau compus). În cazul unui masiv, elementele acestuia erau de același tip: întreg, real, caracterial etc; fără a fi posibilă atribuirea diferitor elemente ale masivului valori de tipuri diferite. Însă deseori apar situații cînd este necesară prelucrarea și păstrarea unei informații mai complexe, așa că orarul lecțiilor, reușita unui student etc. Dacă precăutăm cazul cu reușita studentului, atunci este simplu de presupus că va fi necesară următoarea informație: numele studentului, grupa, notele la examen, balul mediu calculat. Aceste date sînt legate între ele prin faptul că aparțin aceleiași persoane. Ca urmare ar fi justificată tratarea lor ca o singură valoare compusă. Însă tipurile datelor deferă între ele: numele și grupa vor fi de tip șir, notele la examen - de tip întreg, iar balul mediu calculat - de tip real(float). Gruparea acestor componente într-o variabilă compusă este posibilă folosind un tip nou de date numit în limbajul C *structură*.

Primul pas în gruparea componentelor de diferite tipuri într-o variabilă compusă este declararea și descrierea structurii. Declarînd o structură, se creează un tip nou de date a utilizatorului, care pînă în momentul dat n-a fost cunoscut de compilator. Declararea structurilor se face în partea declarării tipurilor, înainte de începutul funcției principale main().

Declararea unei structuri începe cu cuvîntul chee "*struct*", după care urmează numele structurii, care se mai numește tip înregistrare. Elementele unei variabile de tip înregistrare sînt înscrise după numele structurii între acolade. Sintaxa de descriere a elementelor structurii e analogică cu sintaxa declarării variabilelor: se indică numele și tipul elementelor din structură despărțite prin simbolul ";". Descrierea structurii de asemenea se termină cu simbolul ";". Sintaxa descrierii unei structuri în caz general este următoarea: *struct nume {tip_1 nume_1; tip_2 nume_2;.....; tip_n nume_n;};*

Lista elementelor din structură poartă numele de șablon. O structură, ca atare nu declară nici o variabilă. Elementele unei structuri nu sînt variabile aparte, ele sînt componente ale unei sau a mai multor variabile. Astfel de variabile se numesc structurale și trebuie declarate ca fiind de tipul structurii respective. Șablonul respectiv va descrie aceste componente, astfel va fi determinat volumul de memorie necesar de rezervat pentru fiecare variabilă structurată de tip înregistrare.

Dacă precăutăm exemplu cu reușita unui student, declararea unei structuri va fi următoarea : *struct stud {char name [20] ; int ex1, ex2; float med; char grup [10];};*

9.1. Declararea variabilelor de tip structură.

Declararea structurii nu rezervează spațiu de memorii pentru ea. Înainte de a folosi orice tip de date, e necesar de declarat o variabilă corespunzătoare. Este imposibil de a folosi o structură în program fără a declara o variabilă de tip înregistrare, fix așa cum e imposibilă folosirea unei valori de tip float înaintea declarării variabilei de tip float.

Sintaxa de declararea a unei variabile-structuri e următoarea:

```
struct nume_structura nume_variabilă;
```

Aici cuvântul cheie *struct* indică compilatorului faptul că merge vorba despre o structură, iar tipul înregistrării *stud* determină șablonul după care va fi compusă variabila.

După tipul înregistrării urmează numele variabilei, care va fi folosit în program.

De exemplu, pentru a primi acces la datele despre reușita unui student trebuie declarată o variabilă: *struct stud a;* Acum avem variabila *a* compusă din 5 câmpuri, pentru care a fost rezervată memorie.

Dacă în program trebuie folosite câteva variabile de unul și același tip înregistrare, e posibil de folosit următoarea sintaxă: *struct stud a, b, c;* Aici sînt declarate 3 variabile de tip înregistrare *stud*: *a, b, c*. În cazul când e necesară folosirea variabililor de tip înregistrare diferit, ele se vor declara aparte.

Există posibilitatea declarării variabilei de tip înregistrare odată cu descrierea structurii. Pentru aceasta numele variabilei va fi amplasat între acolada de închidere și simbol “;” la sfârșitul declarării structurii. Exemplu:

```
struct stud { char name [20];  
              char grup [10] ;  
              int ex1, ex2;  
              float med;}a;
```

Aici *stud* este tipul înregistrării și numele unui nou tip de date numit structură. Elementele, din care se compune structura se mai numesc câmpuri structurate, *a* este numele variabilei, care va fi folosită în program și e compusă conform șablonului din 5 componente.

9.2. Inițierea variabilelor tip înregistrare.

În cazul, când valorile inițiale ale componentelor unei variabile-structură sînt cunoscute, este posibilă atribuirea lor în timpul declarării variabilei. În cazul când se declară o simplă variabilă tip înregistrare inițierea va fi o parte a declarării structurii:

```
struct stud { char name [20]; char grup [10];  
              int ex1, ex2; float med;}
```

a={“Ivanov”, “SOE-991”, 8,7,7.5};

Aici a fost descrisă structura *stud* și concomitent declarată variabila *a* cu inițializarea valorilor pentru componentele sale.

O altă variantă de inițializare a componentelor structurii este inițializarea lor în tipul declarării unei variabile tip înregistrare. Exemplu:

```
struct stud { char name [20]; char grup [10];  
    int ex1, ex2; float med;}  
main () { struct stud a={“Ivanov”, “SOE-991”, 8,7,7.5}; }
```

O structură este globală, dacă e declarată înainte de funcția principală *main()*, și este locală dacă e declarată înăuntru funcției *main()*, sau înăuntru altei funcții. Însă dacă e necesară inițializarea unei structuri ce conține șiruri, ea trebuie declarată înainte de funcția *main()* sau ca variabilă statică: *static struct stud;*

9.3. Folosirea structurilor.

O structură poate fi prelucrată în program, numai dacă există declarată o variabilă de tip înregistrare. Această variabilă e compusă din câteva elemente fiecare avînd valoare de tip diferit. Însă adresarea către valorile ce se conțin în câmpurile structurii este imposibilă folosind nemijlocit numele câmpului respectiv. De asemenea e imposibilă accesarea valorilor din variabila tip înregistrare folosind numai numele ei. Conform regulilor de sintaxă a limbajului C++, pentru a accesa un element al structurii, este necesar de indicat numele variabilei tip înregistrare și numele câmpului respectiv folosind următoarea sintaxă: *nume_var.nume_cîmp* , unde *nume_var* este numele variabilei tip înregistrare, iar *nume_cîmp* – numele câmpului respectiv din variabilă.

Notă: La prelucrarea valorilor din câmpurile structurii se vor folosi toate funcțiile, instrucțiunile și operațiile aplicabile tipului cărui îi aparțin câmpurile structurii.

Exemplu: De alcătuit o structură care ar conține informații despre un student (nume, grupa) și ar calcula nota medie după rezultatele la două examene:

```
struct stud {char name [20], group[10];  
    int ex1, ex2; float med;};  
void main (void) { struct stud a;  
    puts(“Culege numele și grupa”);  
    gets(a.name); gets(a.group);  
    puts(“Culege notele la două examene”);  
    scanf(“%d%d”,&a.ex1,&a.ex2);  
    a.med=(a.ex1+a.ex2)/2;  
    printf(“media=%f”,a.med);}
```

9.4. Structuri imbricate.

Tipul structură este un tip compus, aceasta înseamnă că o variabilă de acest tip poate fi alcătuită din câteva elemente simple sau compuse, care la rândul său pot deasemenea să conțină alte elemente. Acest fapt ne dă posibilitate să folosim unele structuri în calitate de câmpuri pentru altele structuri. Astfel de structuri se vor numi imbricate. Cantitatea de niveluri de structuri imbricate teoretic poate fi infinită, însă nu se recomandă de a folosi foarte multe nivele de imbricare din cauza sintaxei incomode.

În cazul când structura A conține în componența sa un câmp, care la rândul său este deasemenea o structură B, atunci structura A trebuie declarată numai după ce va fi declarată structura B. Următorul exemplu folosește variabila *a* de tip structură *stud* imbricată, care conține informația despre un student și rezultatele unei sesiuni. Informația despre rezultatele sesiunii este grupată într-o structură aparte cu numele *sesia* și e folosită în calitate de câmp *nota* în structura *stud*:

```
struct sesia {int ex1,ex2,ex3;
float med;};
struct stud {char name [20], group[10];
struct sesia nota;};
void main (void) { struct stud a;
puts("Culege numele și grupa");
gets(a.name); gets(a.group);
puts("Culege notele la 3 examene");
scanf("%d%d%d",&a.nota.ex1, &a.nota.ex2, &a.nota.ex3);
a.nota.med=( a.nota.ex1+a.nota.ex2+a.nota.ex3)/3;
printf("\nmedia=%f",a.nota.med);}
```

Aici a fost calculată nota medie reieșind din rezultatele la 3 examene. Din acest exemplu este simplu de observat sintaxa de adresare la o valoare dintr-un câmp a unei structuri imbricate: se folosesc numele fiecărui câmp imbricat, separat prin punct și scrise în ordine descrescătoare pînă la câmpul destinație. Sintaxa *a.nota.med* înseamnă ca se face o adresare la câmpul *med* ce aparține unei structuri (*sesia*), care la rândul său este câmp (*nota*) în componența altei structuri (*stud*) apelată prin intermediul variabilei *a*.

9.5. Masive de structuri.

În timpul declarării unei variabile tip înregistrare, în memorie se înregistrează spațiu pentru păstrarea și prelucrarea datelor ce se vor conține în câmpurile structurii

conform șablonului numai pentru o înscriere: un student, o persoană ș.a.m.d. În cele mai dese cazuri este necesară prelucrarea informației despre un grup de persoane, în cazul nostru despre un grup de studenți. În acest caz este necesară declararea a mai multor variabile de tip înregistrare, fiecare reprezentînd înscrierea concretă, pentru un student aparte.

Pentru a sistematiza informația păstrată într-o mulțime de înscrieri este binevenită declararea unui masiv de structuri. În acest caz fiecare element al masivului unidimensional tip structură va păstra informația despre o persoană și numărul maximal persoane înregistrate în așa fel va fi egal cu cantitatea de elemente ce se conțin din masiv.

Un masiv unidimensional de structuri se poate declara în modul următor:

struct nume_structură nume_masiv [N]; unde N - numărul de elemente din masiv. Exemplu: *struct stud x[10];* În așa fel au fost rezervate 10 regiuni de memorie, fiecare avînd volumul necesar pentru a păstra o structură întreagă pentru prelucrarea informației despre reușita unui student.

Adresarea în program la un element al masivului de structuri va fi făcută ca și în cazul unui masiv simplu: se va indica numele masivului cu numărul de ordine al elementului în paranteze pătrate. Exemplu: *x[3]*. Însă așa o adresare către structură va fi încorectă. Prelucrarea structurii are loc prin intermediul prelucrării cîmpurilor aparte din ea. Accesarea unui cîmp din variabila de tip înregistrare care este în acelaș timp cîmp al masivului va fi posibilă folosind sintaxa: *nume_masiv[k].Nume_cîmp*, unde *k* – este numărul de ordine a înregistrării necesare.

Exemplu: Pentru a înscrie în cîmpul “*name*” din structura “*stud*” numele studentului cu numărul de ordine 2 vom folosi următoarea sintaxă: *struct stud x[10]; gets(x[2].name)*.

Notă: De nu uitat că numărarea elementelor în masiv se face începînd cu indexul zero. În cazul nostru fiind alcătuit un masiv din 10 structuri , numărul de ordine va varia de la 0 la 9.

Exemplu: Este dată o bază de date cu n înscrieri ce conțin informația despre reușita unui grup de studenți. După notele date la examenele unei sesiuni de calculat nota medie pentru fiecare student. Cîmpuri necesare: Numele, Grupa, Notele la examene, Balul mediu calculat:

```
#include<conio.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
struct stud {char name[20], grupa[10];
```



```

int ex1,ex2; float med;};
void main (void){ clrscr();
struct stud x[50]; int i,N;
printf("Culege numarul studentilor\n");
scanf("%d",&N);
printf("Culege informatia despre %d studenti:\n",N);
for (i=0; i<N; i++){
printf("Culege numele studentului %d\n",i);scanf("%s",x[i].name);
printf("Culege grupa studentului %d\n",i); scanf("%s",x[i].grupa);
printf("Culege 2 note pentru studentul %d\n",i);
scanf ("%d%d",&x[i].ex1,&x[i].ex2);
x[i].med=(x[i].ex1+x[i].ex2)/2;}
printf("\nMasivul final este:\n");
printf("*****\n");
printf("*** Numele ** Grupa ** nota1 ** nota2 ** media **\n");
printf("*****\n");
for (i=0;i<N;i++) {
printf("***%10s**%9s**%7d**%7d**%9.2f**\n",
x[i].name,x[i].grupa,x[i].ex1,x[i].ex2,x[i].med);
printf("*****\n");
}
getch(); }

```

10. Funcții în C/C++.

Rezolvarea eficientă a problemelor cu ajutorul tehnicii de calcul presupune folosirea tuturor posibilităților și instrumentelor limbajului în care are loc programarea. În timpul studierii instrucțiunilor și tipurilor noi în C/C++ apar posibilități de rezolvare eficientă a unor probleme complicate. Însă odată cu creșterea complicității problemei de rezolvat, se mărește volumul programului și complicitatea lui. În acest caz există necesitatea de a evidenția careva sarcini concrete din program împărțindu-l în module separate numite funcții.

Fiecare funcție din program trebuie să îndeplinească o singură sarcină. De exemplu, dacă se află notele medii la un grup de studenți, atunci se poate de creat 3 funcții: prima pentru culegerea valorilor inițiale despre studenți; a doua - pentru calcularea notei medii și a treia - pentru afișarea rezultatelor. Folosind aceste funcții, dacă în program trebuie de îndeplinit o sarcină oarecare, atunci se apelează la funcția

respectivă asigurînd-o cu informația necesară pentru prelucrare. Folosirea funcțiilor în C++ presupune respectarea următoarelor concepții de bază:

- a) Funcțiile grupează setul de operatori pentru îndeplinirea unei sarcini concrete.
- b) Programul principal apelează la funcție, adresîndu-se la numele ei, după care urmează paranteze rotunde. Exemplu: *afișare()*.
- c) După terminarea prelucrării informației, majoritatea funcțiilor întorc programului principal valori de tipuri concrete. De exemplu: *int* sau *float*, care pot fi folosite în calcule.
- d) Programul principal transmite funcțiilor parametrul (informația inițială), inclusă în paranteze rotunde, care urmează după numele funcției.
- e) Limbajul C++ folosește prototipi de funcție pentru determinarea tipului valorii returnate de către funcție, deasemenea a cantității și tipurilor parametrilor transmiși funcției.

Odată cu mărirea volumului și complicității programului, folosirea funcțiilor va deveni condiția principală pentru rezolvarea eficientă și corectă. În același moment crearea și folosirea funcțiilor sînt proceduri simple.

În timpul creării programului e necesar de rezervat fiecare funcție pentru rezolvarea unei sarcini. Dacă apar situații, cînd funcția rezolvă cîteva sarcini, ea trebuie divizată în cîteva funcții mai mici. Fiecare funcție creată trebuie să primească un nume unic. Ca și în cazul cu variabilele, numele unei funcții este un identificator și e de dorit să corespundă cu sensul logic al sarcinei pe care o îndeplinește.

Funcțiile în C++ se aseamănă la structură cu funcția principală *main()*. În fața numelui funcției se indică tipul ei, iar după numele funcției urmează lista de parametri descriși înăuntrul parantezelor rotunde. Corpul funcției compus din operatori este amplasat după descrierea parametrilor și-i înconjurat cu acolade deschise și închise. Sintaxa descrierii unei funcții este următoarea:

tip_f nume_f (lista parametri) {declarare de variabile; operatori;}

unde *tip_f* este tipul funcției sau tipul valorii returnate de funcție, *nume_f* este numele funcției. Dacă facem analogie dintre funcție și programul principal *main()*, atunci putem scrie: *void main (void) {corpul programului}*, unde funcția nu întoarce rezultate (cuvîntul *void* înaintea funcției *main()*) și nu primește parametri din exterior (cuvîntul *void* între paranteze rotunde după funcția *main()*).

Următorii operatori determină funcția cu numele *afișare()*, care afișază la monitor un mesaj: *void afișare (void) {printf("Hello World\n");}*

După cum a fost spus, cuvîntul *void* ce precedează numele *afișare* indică funcției că nu trebuie de întors vre-o valoare în program, iar cuvîntul *void* după

numele afișare indică (compilatorului C++ și programatorului ce citește acest cod) că funcția nu folosește parametri (informație inițială pentru îndeplinirea sarcinei).

Următorul program folosește funcția *afișare()* pentru afișarea mesajului pe ecran:

```
#include<stdio.h>
#include<conio.h>
void afisare (void) {
printf("\n Hello World"\n);
void main (void) {
puts ("inaintea folosirii funcției");
afisare();
puts ("dupa folosirea funcției");
getch(); }
```

Ca și orice program în C++ acest exemplu va fi îndeplinit începînd de la funcția principală *main()*. Înăuntru programului operatorul de apel la funcție apelează la funcția de afișare astfel: *afișare()*; unde parantezele rotunde după identificator indică compilatorului că în program se folosește funcția *afișare()*. Cînd programul va întîlni apelul la funcție, se va începe îndeplinirea tuturor operatorilor din corpul funcției, și după aceasta executarea programului principal va fi continuată cu operatorul amplasat nemijlocit după operatorul de apel la funcție. Următorul program-exemplu conține două funcții, prima afișază pe ecran o salutare, iar a doua tema lecției:

```
#include<stdio.h>
#include<conio.h>
void hello (void){
printf("\n Hello\n");}
void tema (void){
printf("Funcții in C++\n");}
void main (void){
hello();
tema(); }
```

În rezultatul îndeplinirii acestui exemplu vor fi afișate pe ecran două mesaje: "Hello" și "Funcții în C++ ", în ordinea cum apar în program.

Funcțiile prezentate mai sus îndeplinesc sarcini foarte simple. În aceste cazuri programul putea fi alcătuit fără folosirea funcțiilor, cu includerea acelorași operatori în corpul funcției *main()*. Însă funcțiile au fost folosite pentru a analiza descrierea și folosirea ulterioară a funcțiilor. În timpul rezolvării problemelor complicate va fi posibil, în așa fel, de simplificat rezolvarea, împărțind sarcina în module aparte realizate de funcții. În acest caz va fi simplu de observat că analiza și modificarea

funcțiilor este cu mult mai simplă decât prelucrarea unui program voluminos și complicat. Mai mult ca atât, funcția creată pentru un program poate fi folosită și în alt program fără schimbări. În așa mod pot fi alcătuite biblioteci de funcții, folosirea cărora cu mult va micșora timpul folosit pentru alcătuirea programelor.

10.1. Transmiterea parametrilor în funcție.

Pentru a mări posibilitățile funcțiilor din program, limbajul C++ permite de a transmite informație în ele. Informația inițială transmisă din program în funcție la momentul apelului acesteia se numește parametru. Dacă funcția folosește parametri, ei trebuie descriși în timpul descrierii funcției. În timpul descrierii parametrilor funcției se indică numele și tipul fiecărui parametru în următorul mod:

tip_parametru nume_parametru;

Dacă funcția conține câțiva parametri, ei vor fi descriși împreună între parantezele rotunde după numele funcției despărțiți prin virgulă după cum urmează:

*tip_funcție nume_funcție (tip_parametru1 nume_parametru1,
tip_parametru2 nume_parametru2
.....
tip_parametruN nume_parametruN);*

Funcția din exemplu următor folosește un parametru de tip întreg, care în rezultatul îndeplinirii sale, îl afișează la monitor.

```
#include<stdio.h>
void număr (int a) {
printf("Parametru=%d\n",a);}
void main (void) {
număr (1);
număr (17);
număr (-145); }
```

În rezultatul îndeplinirii acestui exemplu vor fi primite 3 fraze ca răspuns:

```
Parametru=1
Parametru=17
Parametru=-145
```

În timpul executării acestui program, funcția *număr()* este apelată de 3 ori cu diferite valori a parametrului. De fiecare dată cînd în program este întâlnită funcția *număr()* valoarea parametrului este înlocuită în funcție și rezultatul este diferit.

Parametrii funcției pot fi de tipuri diferite. În cazul exemplului precedent parametrul a este de tip întreg. Dacă în program va fi făcută încercarea de a

transmite funcției un parametrul de alt tip, de exemplu *float*, compilatorul va întoarce o eroare. Majoritatea funcțiilor folosesc mai mulți parametri, în acest caz trebuie să fie indicat tipul fiecărui parametru. Următorul exemplu folosește funcția *muncitor()*; care afișează numele și valoarea salariului unui muncitor:

```
#include<stdio.h>
void muncitor (char nume[15], float salariu)
{printf("Muncitorul %s are salariu= %f lei\n",nume,salariu);}
void main (void) {
muncitor("Ivanov", 355.35);
muncitor("Petrov", 560.00);} }
```

În rezultatul îndeplinirii acestui program funcția *muncitor()* este apelată de două ori și sînt afișate mesajele:

Muncitorul Ivanov are salariul= 355.35 lei.

Muncitorul Petrov are salariul= 560.00 lei.

Notă: În unele surse de descriere a limbajului de programare C/C++ parametrii ce se transmit din program în funcție se numesc actuali, iar parametrii ce sînt declarați în antetul funcției și cărora le se atribuie valorile parametrilor actuali se numesc parametri formali.

Așadar în exemplul precedent valorile „*Ivanov*” și *355.35* sînt parametri actuali, iar *char num[15]* și *float salariu* sînt parametri formali.

Următorul exemplu folosește funcția *max()*, care compară 2 numere întregi:

```
#include<stdio.h>
void max (int a, int b) {
if (a>b) printf ("%d este mai mare ca %d\n",a,b);
else if (b>a) printf ("%d este mai mare ca %d \n",b,a); }
else printf ("%d este egal cu %d \n",a,b); }
void main (void) {
max(17,21);
max(5,3);
max(10,10); }
```

Deci în timpul folosirii parametrilor în funcție este necesară respectarea următoarelor reguli:

- Dacă funcția folosește parametri, ea trebuie să indice numele unical și tipul fiecărui parametru.
- Când programul apelează la funcție, compilatorul atribuie valoarea parametrilor de la stînga la dreapta.

- Valorile transmise din program în funcție, trebuie să coincidă ca număr, loc și tip cu parametrii din funcție.

10.2. Întoarcerea valorilor din funcție.

Destinația oricărei funcții este îndeplinirea unei sarcini concrete. În majoritatea cazurilor funcțiile vor efectua careva calcule. După aceasta funcția va întoarce rezultatul funcției din care a fost apelată fie aceasta funcția principală *main()*, fie altă funcție. La momentul când funcția întoarce o valoare, trebuie să fie cunoscut tipul ei. Tipul valorii returnate de funcție se indică în timpul descrierii funcției înainte de numele ei. Tipul valorii returnate se mai numește și tipul funcției.

Funcția din următorul exemplu adună 2 numere întregi și întoarce rezultatul programului principal:

```
#include<stdio.h>
#include<conio.h>
int suma (int a, int b) {
    int R;
    R=a+b;
    Return (R);}
void main (void) { int K;
    K=suma(15,8);
    printf("suma=",K);}
```

În acest caz, cuvântul *int*, ce se află înaintea numelui funcției *suma()* în timpul descrierii acestuia, este tipul valorii returnate de funcție în programul principal.

Funcțiile folosesc operatorul *return* pentru a întoarce valori funcțiilor din care au fost apelate. Când compilatorul întâlnește operatorul *return*, el întoarce valoarea dată și încheie executarea funcției curente, controlul executării programului fiind transmis funcției din care a fost chemată funcția curentă. Dacă după operatorul *return*, în funcție mai există operatori, ei vor fi ignorați, funcția terminându-se odată cu îndeplinirea operatorului *return*.

Funcția *suma()* din exemplu precedent este alcătuită din 3 instrucțiuni. Există posibilitatea de a micșora corpul acestei funcții după cum urmează în exemplu următor:

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

În acest caz, în calitate de parametru pentru operatorul *return* a fost folosită o expresie ce calculează suma a două numere. În general, parametrul *return* folosește în calitate de parametrul o expresie ce are rezultate tip identic cu tipul funcției, adică tipul valorii returnate de funcție.

Nu toate funcțiile returnează valori de tip întreg. Următorul exemplu folosește funcția *media()* , care returnează media aritmetică a 3 numere întregi. Evident rezultatul funcției poate fi și un număr real:

```
#include<stdio.h>
#include<conio.h>
float media (int a, int b, int c,){
    return (float(a+b+c)/3.0); }
float R;
R=media(5,7,10);
printf("media este =%f",R); }
```

În rezultatul îndeplinirii acestui exemplu, va fi primit rezultatul R=7,333.... În funcție a fost folosită conversia (*float(a+b+c)*) a unui număr întreg în echivalentul său real pentru a primi ca rezultat media aritmetică a 3 numere de tip real. În acest exemplu cuvântul *float* ce se află înaintea numelui funcției *media()* în momentul descrierii acesteia, indică tipul valorii returnate.

Există situații când operatorul *return* este folosit în funcții ce nu întorc valori.

Următorul exemplu demonstrează această posibilitate:

```
#include<stdio.h>
#include<conio.h>
max (int a, int b) {
    if (a>b) printf("%d>%d\n",a,b);
    else if (a<b) printf("%d<%d\n",a,b);
    else printf ("%d=%d\n",a,b);
    return 0;}
void main (void) {
    max (17,21);
    max (3,-7);
    max (5,5); }
```

Valoarea întoarsă de funcție poate fi folosită în orice loc al programului, unde e posibilă folosirea unei valori de tip identic cu valoarea returnată. Când funcția întoarce o valoare, această valoare poate fi atribuită unei variabile de același tip folosind operatorul de atribuire. Exemplu: *R=media (5,7,10);*

În continuare la acele spuse, însăși numele funcției poate fi folosit, în cazul când există necesitatea folosirii valorii returnate de funcție, de exemplu valoarea returnată poate fi folosită în interiorul funcției de afișare:

```
printf("Media este =%f",media (5,7,10));
```

Mai mult ca atât funcția poate folosi valoarea returnată în condiții, după cum urmează:

```
if (media(5,7,-14)<0) printf("media este negativa\n");
```

10.3. Prototipul funcției.

Înainte de a face apelul unei funcții compilatorul C++ trebuie să cunoască tipul valorii returnate, cantitatea și tipul parametrilor folosiți de funcție. În fiecare exemplu precăutat pînă în prezent descrierea funcției era făcută înaintea apelului ei din programul principal. Însă sînt situații, cînd unele funcții în program sînt apelate reciproc. În așa cazuri este posibilă situația cînd o funcție va fi apelată înainte descrierii sale.

Pentru a garanta faptul, că compilatorul C++ cunoaște particularitățile fiecărei funcții folosite în program, se folosesc prototipi ai funcțiilor. Prototipul unei funcții este amplasat la începutul programului și conține informația despre tipul valorii returnate, cantitatea și tipul parametrilor folosiți de funcție.

Exemplul următor demonstrează cum s-ar putea alcătui prototipi pentru funcțiile folosite anterior.

```
void afișare (void);  
void hello (void);  
void tema (void);  
void număr (int);  
void muncitor (char, float);  
void max (int, int);  
int suma (int, int);  
float media (int, int, int);
```

După cum se vede din exemplu fiecare funcție își are prototipul său care descrie tipul valorii returnate, cantitatea și tipul parametrilor. Este important de nu uitat de simbolul ”;” la sfîrșitul fiecărui prototip. Lipsa lui duce la eroare în compilare, la fel cum duce la eroare lipsa prototipului unei funcții dacă aceasta apare în program înaintea descrierii sale. Fiind declarat prototipul unei funcții înainte de a fi început corpul programului, descrierea ei poate fi făcută după acolada de închidere a programului principal. Exemplu:

```
#include<stdio.h>  
#include<conio.h>  
float media (int,int,int);  
void main (void) {float r;  
clrscr(); r=media (5,17,10);
```



```
printf ("media=%f/n",r);
getch (); }
float media (int a, int b, int c) {
return (float(a+b+c)/3.0);}

```

10.4. Variabile locale și domeniul de vizibilitate.

Funcțiile folosite în exemplele precedente îndeplinesc niște sarcini relativ simple. La momentul, când funcțiile vor trebui să îndeplinească niște sarcini mai complicate, va apărea necesitatea folosirii în funcții a variabilelor proprii.

Variabilele declarate în cadrul funcției se numesc locale. Numele și valoarea unei variabile locale sînt cunoscute numai funcției în care ea a fost declarată. Chiar faptul că variabila locală există este cunoscut numai funcției în care ea a fost declarată. Declararea variabilelor are loc la începutul funcției, îndată după acolada ce deschide corpul acesteia. Numele variabilei locale trebuie să fie unicul numai în funcția în care a fost declarată. O variabilă se numește locală, din cauză că este văzută numai din funcția în care a fost descrisă. Sintaxa de declarare a unei variabile locale este:

```
tip_f numele_f (lista parametrilor) {tip_vl numele_vl;}
unde: tip_f - tipul funcției; nume_f - numele funcției; tip_vl - tipul variabilei;
numele_vl - numele variabilei;
```

Principiile de declarare și folosire a unei variabile locale oricărei funcții sînt identice cu principiile de declarare și utilizare a unei variabile declarate în corpul funcției principale *main()*; O variabilă declarată în corpul funcției *main()* este locală acestei funcții.

În general, tot ceia ce a fost spus despre variabilele declarabile în funcția *main()*: tipurile, numele, principiile de utilizare ș.a. este aplicabil pentru o variabilă locală din orice altă funcție.

Următorul exemplu folosește funcția *fact()* pentru calcularea factorialului unui număr.

```
#include<stdio.h>
#include<conio.h>
int fact (int R) {
int i, k=1;
for (i=1;i<=R;i++){k*=i;}
return (k);}
void main (void) {
int n; clrscr();
```

```
printf("Culege o cifra\n");
scanf("%d",&n);
printf("%d!=%d\n",n,fact(n));
getch();}
```

În exemplu precedent funcția *fact()* folosește 2 variabile locale *i* și *k* de tip întreg. Variabila *k* se inițializează cu valoarea 1 chiar în momentul declarării sale. Asume în această variabilă va fi păstrată valoarea factorialului pe parcursul execuției funcției *fact()*.

Să analizăm acest exemplu. La începutul executării programului, utilizatorul culege o cifră de tip întreg din care mai apoi va fi calculat factorialul. Cifra culeasă de utilizator se păstrează inițial în variabila *n*, apoi odată cu apelul funcției *fact()* se transmite ca parametru fiind atribuită parametrului formal *R* din funcție. La începutul executării funcției, variabilei locale *k*, în care va fi păstrată valoarea factorialului *i* se atribuie valoarea inițială 1 pentru a evita orice valori întâmplătoare. Asume prin intermediul variabilei locale *k* funcția întoarce valoarea finală a factorialului în programul principal. În continuare ciclul *for* e repetat de *n* ori pentru a calcula valoarea factorialului *n!*, înmulțind de fiecare dată valoarea nouă a parametrului ciclului *i* la valoarea veche a lui *k*. La sfârșit valoarea finală a lui *k* va fi întoarsă în program cu ajutorul operatorului *return ()*.

În timpul declarării variabilei locale pentru funcții există probabilitatea, că numele variabilei locale declarate într-o funcție să fie identic cu numele variabilei locale din altă funcție. După cum a fost spus, o variabilă locală este cunoscută numai în funcția, în care a fost declarată. Așadar, dacă două funcții folosesc același nume pentru variabilele sale locale, aceasta nu aduce la situație de conflict. Compilatorul C++ precaută numele fiecărei variabile ca locală pentru funcția corespunzătoare.

Următorul exemplu folosește funcția *suma()* pentru adunarea a două numere întregi. Această funcție atribuie rezultatul îndeplinirii sale variabilei locale *x*. Însă funcția *main()* folosește în calitate de parametru transmis funcției *suma()* deasemenea o variabilă cu numele *x*. Din cele spuse mai sus rezultă că ambele variabile vor fi precautate ca locale și nu va apărea situație de conflict.

```
#include <stdio.h>
#include<conio.h>
int suma(int a, int b) {
    int x;
    x=a+b;
    return(x); }
void main (void) {
```

```

int x,y;
printf(„Culege 2 cifre\n”);
scanf(„%d%d”,&x,&y);
printf(„%d+%d=%d”,x,y,suma(x,y));
getch(); }

```

Exemplul următor folosește funcția *suma()* pentru a calcula suma elementelor unui masiv unidimensional de tip întreg. În calitate de parametri se folosesc masivul și mărimea sa.

```

#include<stdio.h>
#include<conio.h>
int suma (int y[10], int m) {
    int i, sum=0;
    for (i=0;i<m;i++){
        sum+=y[i];}
    return(sum); }
void main (void) {
    int w,n,i,x[10]; clrscr();
    printf(“Culege marimea masivului n<10\n”);
    scanf(“%d”, &n);
    for (i=0;i<n;i++){
        printf(„Culege elementul x[%d]\n”,i);
        scanf(“%d”,&x[i]); }
    w=suma(x,n);
    printf(“suma masivului = %d\n”,w);
    getch(); }

```

10.5. Variabile globale.

Numim variabilă globală o variabilă, numele și valoarea căreia sînt cunoscute pe parcursul întregului program. Despre existența unei variabile globale într-un program C++ știe orice funcție din acest program. Pentru a crea o variabilă globală se folosește declararea ei la începutul programului în afara oricărei funcții. Orice funcție, care va urma după așa o declarare poate folosi această variabilă globală. Declararea unei variabile globale are următoarea sintaxă:

```

#include<stdio.h>
tip_vg nume_vg;
void main (void)
{ ... }

```

unde *tip_vg* este tipul variabilei globale, iar *nume_vg* – numele variabilei globale.

Fiind declarată o variabilă globală, valoarea ei nu numai e cunoscută oricărei funcții din program, dar și poate fi schimbată din oricare din funcțiile prezente în program.

Exemplu următor folosește variabila cu numele *cifra*. Fiind accesibilă din oricare din 2 funcții prezente în program, valoarea variabilei globale *cifra* este schimbată pe rînd din ambele funcții:

```
#include<stdio.h>
int cifra=100;
void f1 (void) {
printf(“cifra=%d\n”, cifra);
cifra*=2; }
void f2 (void) {
printf(“cifra=%d\n”,cifra);
cifra+=2;}
void main (void) {
printf(“cifra= %d\n”, cifra);    //100
cifra ++;
f1();                          //101, 202
f2 ();                          //202, 204
printf(“cifra=%d\n”,cifra);    //204
getch(); }
```

Cu toate că prezența variabilelor globale în program adaugă noi posibilități, e de dorit de a evita folosirea lor frecventă. Din cauza, că orice funcție din program poate schimba valoarea variabilei globale este foarte greu de urmărit toate funcțiile, care ar putea schimba această valoare, ceea ce duce la control dificil asupra execuției programului. Pentru a rezolva această problemă se poate declara variabila în corpul funcției *main()* și apoi de a o transmite altor funcții, în calitate de parametru. În acest caz în stivă va fi amplasată copia temporară a acestei variabile, iar valoarea inițială (originalul) va rămâne neschimbată.

10.6. Conflicte dintre variabile locale și globale.

În cazul cînd un program trebuie să folosească o variabilă globală, poate apărea situație de conflict, dintre numele variabilei globale și numele variabilei locale. În așa cazuri limbajul C++ oferă prioritate variabilei locale. Adică, dacă există variabilă globală cu același nume ca și variabila locală, compilatorul consideră, că orice apel a variabilei cu așa nume este un apel al variabilei locale. Însă apar situații, cînd există

necesitatea de a se adresa la o variabilă globală ce se află în conflict cu o variabilă locală. În acest caz se poate folosi operatorul global de acces (::).

Următorul exemplu folosește variabila globală *num*. Pe lângă această funcția *afisare()* folosește variabila locală *num*. Folosind operatorul global de acces (::) funcția apelează la variabila globală *num* fără conflict cu variabila locală *num*:

```
#include<stdio.h>
int num=505;
void afisare (void) {
    int num=37;
    printf("variabila locala num=%d\n",num);
    printf("variabila globala num=%d\n",::num);}
void main (void) { afisare(); }
```

11. Indicatori (pointeri).

Un program în limbajul de programe C/C++ conține în cele mai frecvente cazuri o mulțime de variabile împreună cu descrierea lor. Fie variabila *x* descrisă în program prin declararea de forma *int x*; În acest caz compilatorul rezervează un spațiu de 2 octeți în memoria calculatorului. La rândul său, celula de memorie unde va fi alocată valoare variabilei *x*, își are adresa sa. De exemplu: 21650. Când variabila *x* primește o valoare, această valoare neapărat se înscrie în celula de memorie rezervată variabilei *x*. Deoarece, în cazul nostru variabila *x* își păstrează valorile pe adresa 21650, celula cu această adresă va conține valoarea atribuită variabilei *x*.

În așa caz, către orice variabilă din program se poate de se adresat în 2 moduri: folosind numele variabilei sau folosind adresa din memorie unde este păstrată valoarea variabilei *x*. Folosind numele variabilei, noi ne adresăm la valoarea variabilei care se păstrează în memorie. Folosind adresa, noi ne adresăm la celula de memorie cu adresa respectivă, unde se păstrează valoarea variabilei.

Limbajul de programare C/C++ conține tehnici mai avansate de lucru cu valoarea unei variabile și adresa de memorie unde se păstrează această valoare. În aceste scopuri se folosesc indicatori. Indicatorul reprezintă o variabilă, ce conține adresa altei variabile. Sintaxa de declarare a unui indicator este următoarea:

*tip *nume;* unde *tip* – este tipul de date a variabilei la care se poate referi indicatorul (adică tipul de date a variabilei, adresa de memorie a căreia va conține indicatorul); astericsul “*” indică că identificatorul ce urmează după el este un indicator.

nume – este un identificator ce desemnează numele variabilei indicator.

Exemple de declarare a indicatorilor:

```

int *x;           // indicator la date de tip întreg.
float *p;         // indicator la date de tip flotant.
char *z;          // indicator la date de tip caracterial.
int *y[3];        // masiv de indicatori la date de tip întreg.
void *k;          //indicator la obiectul datelor tipul cărora nu-i necesar de definit
char *S[5];       // masiv de indicatori la date de tip caracter.
char (*S)[5];     // indicator la date de tip caracter din 5 elemente.

```

Odată ce indicatorul a fost declarat ca referință la date de tip întreg, el nu se va putea referi la o variabilă tip float, cauza fiind volumul diferit rezervat în memorie pentru variabile întregi și flotante. Există indicatori și către elemente fără tip - *void*. Putem atribui unui pointer *void* valoarea altui pointer *non-void*, fără a fi necesară operația de conversie de tip. Exemplu:

```

int *a; void *b;
b=a; //corect
a=b; //incorect, lipsește conversia de tip.

```

Fie că a avut loc o atribuire de forma: *int x=5;*. În acest caz pentru variabila *x* a fost rezervată o zonă de memorie cu volumul de 2 octeți, care își are adresa sa. Adresa zonei de memorie, unde se păstrează valoarea variabilei *x* se poate obține cu operatorul obținerii adresei “&”. Rezultatul operației obținerii adresei este adresa locației de memorie ce a fost alocată pentru variabila respectivă. De exemplu: presupunând că *x* e înscrisă în memorie pe adresa 21650, atunci &*x* va fi egală cu 21650. Este important, că &*x* este constantă de tip indicator și valoarea sa nu se schimbă în timpul execuției programului. Exemplu:

```

#include <stdio.h>
void main (void)
int x=5; float r=1.7;
int *q; float *w;
q=&x; w=&r;
printf (“%f se află pe adresa %d \n”,r,w);
printf (“%d se află pe adresa %d \n”,x,q);}

```

Din exemplu ușor se observă că adresa celulei de memorie se reprezintă printr-o valoare de tip întreg. În același timp această valoare nu poate fi schimbată în program și expresia &*x=55;* este incorectă.

Analizând toate aceste noțiuni, apare întrebarea: “Cu ce scop se folosesc indicatori, dacă valoarea variabilei și valoarea adresei sale se poate păstra în variabile simple?”.

Prioritatea folosirii indicatorului constă în faptul, că la el se poate adresa în 2 moduri: q și $*q$. Astericsul $*$ în acest caz indică că se apelează la conținutul celulei de memorie, adresa căreia este valoarea indicatorului. Adică valoarea variabilei x de tip întreg este 5; valoarea indicatorului q este 21650; iar valoarea lui $*q$ este egală cu cifra de tip întreg 5 înscrisă pe adresa de memorie 21650.

În așa mod:

- 1) Variabila q poate primi valori numai în formă de adresă $q=\&x$ și atribuirea de forma $q=21650$; este incorectă din cauza că aici se încearcă atribuirea unei valori întregi unui indicator și nu a adresei.
- 2) Variabila $*q$ poate primi valori de tip întreg. De exemplu: $*q=6$; Această atribuire se descifrează astfel: de amplasat valoarea întreagă 6 în celula de memorie ce are adresa indicată în variabila q . Din cauza, că variabila q indică la celula cu adresa 21650, valoarea variabilei ce-și păstrează valoarea în celula de memorie cu această adresă va fi egală cu 6. Exemplu:

```
#include <stdio.h>
main void (main){
    int x=5;
    int *q;
    q=&x;
    printf("x=%d\n",x);    // x=5;
    *q=6;
    printf("x=%d\n",x) }    // x= 6;
```

În rezultatul îndeplinirii exemplului vor fi afișate 2 expresii: $x=5$ și $x=6$. Prima valoare primită de variabila x este valoarea 5 atribuită la momentul inițializării acesteia. A doua valoare primită de variabila x este valoarea 6 atribuită celulei de memorie la care indică indicatorul q .

Desigur variabila x ar putea primi valoare nouă prin simpla atribuire, $x=6$; dar efectul deplin de la folosirea indicatorilor se poate observa la transmiterea lor în calitate de parametri unei funcții, la crearea unui fișier, etc.

Este de remarcă faptul, că în anumite cazuri indicatorul neinițializat reprezintă un pericol în program. De exemplu, dacă există necesitatea de a transmite datele pe adresa, ce conține indicatorul, în condițiile când acesta nu e corect inițializat, se poate de șters informația importantă pentru funcționarea sistemului operațional, sau altor programe, fiindcă indicatorul neinițializat poate indica la orice adresă de memorie inclusiv la adresa unde sînt păstrate date importante ale sistemului de operare. Un exemplu de inițializare a indicatorului este: $\text{int } *q=\&x$.

Cu scopul folosirii eficiente a memoriei limbajul C/C++ oferă programatorilor posibilitatea eliberării memoriei alocate, dacă e necesar. Pentru aceasta se folosește funcția *free()*, prototipul căreia se află în biblioteca *alloc.h*. Exemplu:

```
#include <stdio.h>
#include <alloc.h>
main void (main){
    int x=5, *q=&x;
    *q=6;
    printf("x=%d \n", x);    // x=6;
    free (q);
    printf ("x=%d \n", x);    // x=-16;
    printf ("adresa x = %d/n", &x)    // -12.
```

În rezultatul executării acestui program vor fi primite mesajele: *x=6; x=-16, adresa x=-12*. Primul mesaj va fi același, dacă programul va fi executat pe orice calculator. Ultimele 2 mesaje vor fi diferite de la un calculator la altul în dependență de alocarea memoriei. Aici după folosirea funcției *free(q)* valoarea păstrată în celula de memorie la care indică indicatorul *q* va fi pierdută, adică valoarea variabilei *x* va fi pierdută.

11.1. Indicatori și funcții.

Din temele precedente se știe că în timpul lucrului cu o funcție, la apelul ei se poate transmite atâți parametri, de câți este nevoie în program. În același timp cu ajutorul operatorului *return*, se poate întoarce numai o singură valoare din funcție. Și situațiile, când funcția trebuie să calculeze și să întoarcă câteva valori în program par a fi nesoluționate. În calitate de alternativă se pot folosi variabile globale, însă în acest caz deseori se pierde controlul asupra îndeplinirii programului și se complică căutarea greșelilor.

În cazul, când există necesitatea de a întoarce din funcție mai multe valori, este util de folosit indicatori. Fără folosirea indicatorilor, noi trebuie să transmitem parametri funcției după valoare. Aceasta înseamnă, că valoarea parametrului actual se atribuie parametrului formal. Acești parametri ocupă zone de memorie diferite, și deci schimbarea valorii parametrului formal din funcție nu duce la schimbarea valorii parametrului actual.

În cazul transmiterii parametrilor după adresă cu ajutorul indicatorilor, nu se crează copii de valori ce dublează parametrii. În acest caz se crează o a doua variabilă ce indică la aceeași celulă de memorie. În așa fel, dacă este schimbată valoarea

parametrului formal din funcție prin indicator, deasemenea este schimbată și valoarea parametrului actual. Exemplu:

```
#include <stdio.h>
int schimb (int w, int *z)
(*z)*=2; w=w*2;
return(w);}
void main (void)
int x=5, y=8,a;
printf ("x=%d, y=%d",x,y);
a=schimb(x, &y);
printf ("x=%d,y=%d\n",a,y);}
```

În acest exemplu sînt date 2 variabile $x=5$ și $y=8$. Funcția *schimb()* schimbă valorile lor înmulțindu-le la 2. Însă, dacă valoarea lui x este schimbată în funcție și întoarsă în program prin intermediul operatorului *return*, atunci valoarea lui y este schimbată prin intermediul indicatorului. Apelînd funcția *schimb()*, noi transmitem în calitate de parametru nu valoarea variabilei y , ci adresa celulei de memorie, unde ea își păstrează valoarea. În așa fel, orice schimbare a valorii indicatorului $*z$ din funcție va duce la schimbarea valorii variabilei y din program.

Un caz aparte în lucrul cu indicatorii este folosirea indicatorului la un șir de caractere. Șirul de caractere prezintă un masiv tip caracterial. Cînd programul transmite un masiv în funcție, compilatorul transmite adresa primului element al masivului. În rezultat este admisibil ca funcția să folosească indicator la șir de caractere. Exemplul următor calculează de cîte ori într-o frază dată se întîlnește caracterul 'a':

```
#include <stdio.h>
int litera (char *s){
int k=0;
while (*s!='\0') {
printf ("caracterul %c pe adresa %d", *s,s);
if (*s == 'a') k++;
s++;}
return (k);}
void main (void)
int p; char x [25];
puts ("culege o frază");
gets (x);
p=litera(x);
```

```
printf ("fraza conține %d caractere a",p); }
```

Aici, la apelul funcției *litera()* în calitate de parametru se transmite nu adresa șirului de caractere *x*, ci însăși șirul. Aceasta e condiționat de faptul, că la transmiterea masivelor în calitate de parametru actual, se transmite nu masivul întreg, ci adresa primului element al masivului. Deci indicatorul *s* va conține adresa elementelor masivului, iar **s* – valorile lor. În așa mod, la prima iterație a ciclului *while* variabila **s* va avea valoarea egală cu primul caracter din șirul *x* transmis funcției. După îndeplinirea primei iterații funcția mărește valoarea variabilei *s* cu 1 unitate. Aceasta înseamnă că valoarea adresei primului caracter din șir va fi incrementată, în rezultat primindu-se adresa caracterului al doilea din șir. Ciclul se va termina când va fi depistat caracterul nul în șir.

Același lucru se poate de făcut și cu un masiv tip întreg:

```
#include <stdio.h>
#include <conio.h>
int masiv (int *s, int z)
{int k=0, i;
for (i=0; i<z; i++){
printf ("elementul %d=%d si se afla pe adresa %d",i,*s,s);
if (*s==0) k++;
s++;}
return (k);}
void main (void){
int p,n,i, x[25]; clrscr ();
printf ("culege mărimea masivului x\n");
scanf ("%d",&n);
printf (culege masivul x[%d]\n",n);
for(i=0; i<n; i++)
scanf ("%d",&x[i]);
p=masiv (x, n);
printf ("masivul conține %d zerouri",p);
getch(); }
```

În acest exemplu se calculează cantitatea de zerouri din masiv. Mărimea masivului se transmite în funcție de la parametrul actual *n* la parametrul formal *z*. Iar pentru transmiterea masivului *x* în funcție se folosește atribuirea valorii primului element *x[0]* variabilei **s*;

Folosind indicatori se pot alcătui funcții, sarcina cărora ar fi culegerea valorilor elementelor unui masiv unidimensional. Aceste funcții ar fi util de folosit în

programele unde se prelucrează cîteva masive. Următorul exemplu ilustrează acest fapt și face posibil prelucrarea a 2 masive: x și y .

```
#include <stdio.h>
#include <conio.h>
int masiv(int *k){
    int i,z;
    printf("Culege mărimea masivului <50 \n");
    scanf("%d",&z);
    for (i=0;i<n; i++) { printf("culege elementul %d \n",i);
        scanf("%d",k); k++;}
    return (z);}
void main (void){
    int n1, n2, i, x[50], y[50]; clrscr ();
    printf("Întroducem masivul x \n"); n1=masiv(x);
    printf("Întroducem masivul y \n"); n2=masiv(y);
    printf("Afișare masiv x:\n");
    for (i=0; i<n; i++) printf("x[%d]=%d\n",i,x[i]);
    printf("Afișare masiv y:\n");
    for (i=0; i<n; i++) printf("y[%d]= %d\n",i, y[i]);
    getch();}
```

În exemplu precedent elementele masivul și mărimea lui sînt introduse în funcție, iar programul principal afișază rezultatul executării funcției. În calitate de parametri funcția *masiv()* folosește un indicator de tip întreg $*k$, cărui i se transmite din program adresa primului element al masivului prelucrat. În interiorul ciclului valoarea adresei din indicator se incrementează, în așa fel primindu-se adresa următorului element al masivului ce se prelucrează. Funcția întoarce în program valoarea măririi masivului curent ce se atribuie variabilelor $n1$ pentru masivul x și $n2$ pentru masivul y .

Avînd la dispoziție un așa algoritm este ușor de compus programe ce prelucrează mai multe masive și chiar masive bidimensionale. Însă în cazul masivelor bidimensionale trebuie de menționat, că valorile elementelor masivului sînt păstrate în celule de memorie adresele cărora se schimbă liniar pentru elementele masivului pe rînduri apoi pe coloane, adică de la stînga la dreapta și de sus în jos. Exemplu:

```
#include <stdio.h>
#include <conio.h>
int masiv (int(*k)[50]) {
    int i,j,z;
```

```

printf ("Culege marimea masivului\n");
scanf ("%d",&z);
for (i=0; i<z; i++) {
for (j=0; j<z ; j++){
printf ("Culege elementul [%d][%d]\n",i,j);
scanf ("%d",(k[i]+j)); }}
return (z); }

void main (void){ clrscr();
int n,n1,n2,i,j,x[50][50],y[50][50];
printf ("Introducem masivul x\n");
n1=masiv(x);
printf ("Introducem masivul y\n");
n2=masiv(y);
printf ("Masivul x este:\n");
for (i=0; i<n1; i++) {
for (j=0; j<n1; j++) {
printf ("%d ", x[i][j]);}
printf ("\n"); }
printf ("Masivul y este:\n");
for (i=0; i<n2; i++) {
for (j=0; j<n2; j++) {
printf ("%d ", y[i][j]);}
printf ("\n"); }
getch(); }

```

Aici în calitate de parametru funcția folosește un indicator $(*k)[50]$ la masiv unidimensional din 50 elemente. Deci, fiecare schimbare a adresei prin $k++$, va duce la poziționarea pe primul element al următorului rând din masivul bidimensional. Iar fiecare indicator $(k[i]+j)$ în ciclu va duce la poziționare pe elementul coloanei j din rândul i al masivului bidimensional.

12. Fișiere în C/C++.

În timpul executării programelor, în majoritatea cazurilor, există necesitatea afișării rezultatelor. Însă posibilitățile simplei afișări a datelor la monitor sînt destul de mici. Chiar și folosirea opririi temporare a execuției programului, cu scopul de a da posibilitate utilizatorului să citească toată informația afișată, nu rezolvă pînă la sfîrșit problema. Dacă mesajul afișat dispăre din limitele monitorului, afișarea lui repetată va fi posibilă numai după lansarea din nou a programului întreg. Mai mult ca

atît, valorile atribuite variabilelor din program sînt păstrate numai pe parcursul îndeplinirii programului. Odată cu terminarea execuției programului, toată informația introdusă se pierde.

Pentru a păstra informația odată introdusă, cu scopul folosirii ulterioare este necesară înscrierea ei pe disc în structuri de date speciale, ce poartă denumirea de fișiere. Folosirea fișierelor permite păstrarea informației de orice natură pentru un timp îndelungat, transferarea datelor de pe un purtător informațional pe altul, citirea și afișarea datelor în caz de necesitate. Tată informația, în ceea ce privește lucrul cu fișierele în limbajul C/C++ este păstrată în biblioteca `stdio.h`. Deaceea înainte de a începe prelucrarea unui fișier în C/C++, este necesar de inclus această bibliotecă în program cu ajutorul directivei `#include<stdio.h>`, care va face posibilă înscrierea informației în fișier și citirea ei. În timpul intrării datelor în program dintr-un fișier de pe disc, are loc copierea lor în memoria operativă a calculatorului, iar informația din fișier ce se află pe discul rigid rămîne neschimbată pe parcursul executării programului. La ieșirea datelor din program pe disc, în fișier sînt înscrise datele ce se păstrau pînă în acel moment în memoria operativă.

Înscrierea sau citirea informației din fișier este efectuată cu ajutorul indicatorului la fișier. În timpul înscrierii sau citirii informației din fișier compilatorul folosește un nivel intermediar de legătură între program și discul rigid, unde este păstrat fișierul. Acest nivel reprezintă o zonă de memorie numită zona tampon care are destinația de păstrare temporară a informației cu scopul de a o înscrie sau citi apoi din fișier.

Pentru a trimite sau a citi informația din zona tampon compilatorul folosește o structură specială numită structură-fișier. În această structură este păstrată informația necesară calculatorului pentru a efectua înscrierea/citirea datelor din fișier, inclusiv adresa zonei de memorie, unde este amplasat fișierul.

Sintaxa de declarare a unui indicator de fișier este următoarea:

*FILE *file_pointer;*

unde cuvîntul cheie *FILE* indică compilatorului că variabila declarată este un indicator la fișier, iar *file_pointer* este numele indicatorului. În caz cînd programul presupune lucrul cu cîteva fișiere, este necesară declararea a mai multor indicatori la fișiere după cum urmează:

*FILE *f1, *f2, *f3;*

unde *f1, f2, f3* sînt numele indicatorilor la fișiere diferite.

Prelucrarea fișierelor în limbajul C/C++ presupune îndeplinirea următorilor pași:

- 1) Deschiderea fișierului. (Pentru a putea fi posibilă înscrierea sau citirea informației din fișier, el trebuie deschis.)
- 2) Prelucrarea fișierului (operațiunile de citire/înscriere).
- 3) Închiderea fișierului. Pentru ca informația înscrisă în fișier să fie păstrată, acesta trebuie închis.

12.1. Deschiderea fișierelor.

Deschiderea unui fișier se realizează cu ajutorul funcției *fopen()* care are următoarea sintaxă:

pointer=fopen("nume_f", "mod");

unde *pointer* este numele indicatorului la fișier, *nume_f* este numele fișierului real de pe discul rigid, *mod* este modul de acces la fișier.

Rezultatul îndeplinirii acestei funcții este atribuirea adresei structuri tip fișier indicatorului la fișier. Primul parametru este numele fișierului, care de obicei are următoarea structură: *nume.ext*, unde *ext* este extensia fișierului compusă din 3 caractere. În calitate de parametru doi funcția primește modalitatea de acces la fișier, adică informația despre operațiile ce pot fi efectuate cu fișierul.

Există 3 modalități generale de deschidere a fișierului:

- 1) Modul "*w*" permite deschiderea fișierului cu scopul înscrierii în el a informației sau scoaterea informației la imprimantă. Dacă fișierul indicat nu există, el va fi creat. Dacă fișierul deja există, toată informația existentă în el va fi nimicită.
- 2) Modul "*r*" indică compilatorului, că fișierul va fi deschis cu scopul citirii din el a informației. Dacă fișierul nu există la momentul deschiderii va fi generată o eroare de execuție.
- 3) Modul "*a*" permite deschiderea fișierului cu scopul completării, adică a înscrierii informației la sfârșitul lui. În caz că fișierul nu există, el va fi creat din nou. Dacă fișierul indicat există, atunci informația înscrisă va fi amplasată la sfârșitul fișierului fără a fi nimicită informația deja existentă în fișier.

De exemplu, pentru a crea un fișier nou cu nume *info.txt* va fi folosită următoarea sintaxă:

*FILE *fișier;*

fișier=fopen("info.txt", "w");

În cazul când există necesitatea de a citi careva date din fișier trebuie folosit modul de acces "*r*":

*FILE *fișier;*

fișier=fopen("info.txt", "r");

Pentru a imprima informația din fișier pe hîrtie cu ajutorul imprimantei, numele fișierului va fi *PRN*, iar modul de acces “*w*”:

```
FILE *fișier;
```

```
fișier=fopen(“PRN”, ”w”);
```

E de menționat faptul, că și numele fișierului și caracterul ce determină modul de acces la fișier sînt delimitate de ghilimele duble. Aceasta este condiționat de faptul, că parametrii în funcția *fopen()* se transmit în calitate de șiruri de caractere. Folosind aceste posibilități, se poate culege de la tastatură numele fișierului dorit de utilizator:

```
char name [12];
```

```
FILE *f;
```

```
printf (“Culege numele fișierului\n”);
```

```
gets (name);
```

```
f=fopen (name, ”w”);
```

În timpul lucrului cu fișierela în C/C++ este folosit un indicator special, în care se păstrează informația despre poziția curentă de citire din fișier. În timpul citirii a datelor din fișier indicatorul determină porția următoare de date care trebuie citită de pe disc.

Dacă fișierul e deschis pentru prima dată cu regimul de acces “*r*”, indicatorul se amplasează pe primul simbol din fișier. În timpul îndeplinirii operației următoare de citire, indicatorul se amplasează la începutul următoarei porții de date. Mărimea pasului de citire din fișier depinde de cantitatea de informație citită din fișier. Dacă dintr-un pas se citește numai un caracter, atunci indicatorul se va amplasa pe caracterul următor, dacă se citește o structură, indicatorul va fi amplasat pe următoarea structură. La momentul, cînd toată informația a fost citită din fișier, indicatorul nimerește pe un cod special numit sfîrșit de fișier: (*eof*). Încercarea de citire din fișier după sfîrșitul fișierului duce la eroare.

Dacă fișierul este deschis cu regimul de acces “*w*”, indicatorul deasemenea se amplasează la începutul fișierului în așa fel primele date înscrise în fișier vor fi amplasate la începutul lui. În timpul închiderii fișierului, la sfîrșitul lui va fi înscris simbolul de sfîrșit de fișier (*eof*). Dacă la momentul deschiderii în regimul de acces “*w*” fișierul există, toată informația deja existentă în el va fi nimicită și deasupra ei va fi înscrisă informația nouă. Orice date precedente ce pot rămîne neșterse în fișier vor fi amplasate după codul sfîrșit de fișier (*eof*), deci accesul la ele va fi închis. În așa fel toată informația din fișierul deschis în regim de acces “*w*” va fi nimicită, chiar și în cazul cînd fișierul va fi deschis fără înscrierea în el a careva date.

Dacă fișierul se deschide în regim de acces “a”, indicatorul se amplasează pe simbolul sfârșit de fișier (*eof*). Orice informație în așa fel înscrisă în fișier va fi amplasată după datele deja existente, iar după înscriere la sfârșitul fișierului se adaugă codul sfârșit de fișier (*eof*).

În unele cazuri apare situația când sistemul operațional nu poate deschide fișierul indicat în funcția *fopen()*. Aceasta poate condiționat de lipsa de loc pe discul rigid, sau de faptul că fișierul indicat pur și simplu nu există. Este posibilă deasemenea situația când e necesară imprimarea datelor la imprimantă, iar aceasta nu-i inclusă sau lipsește hîrtie. În cazul încercării folosirii fișierului ce nu poate fi deschis, programul va fi stopat în rezultatul unei greșeli de execuție. Pentru a evita stoparea avariata a programului poate fi controlată starea de deschidere a fișierului cu ajutorul instrucțiunii condiționate *if*, care va opri programul în caz de eșec. Aici va fi folosită particularitatea sistemului operațional, care întoarce valoarea *NULL* în caz când apare greșală și fișierul nu poate fi deschis. În acest caz codul *NULL* este întors în loc de adresa structurii-fișier și programul se stopează. Condiția pentru evitarea ieșirii avariate din program va avea următoarea sintaxă:

```
if ( (fișier=fopen("info.txt", "w"))==NULL )  
puts ("Fișierul nu poate fi deschis");  
exit();
```

După ce toată informația este înscrisă în fișier sau citită din el e necesar de închis fișierul, adică de a întrerupe legătura dintre fișier și program. Închiderea fișierului se realizează cu ajutorul funcției *fclose()* care are următoarea sintaxă:

```
fclose(f_pointer); unde f_pointer este numele indicatorului la fișier.
```

Odată cu închiderea fișierului, noi primim garanția că toată informația din zona tampon într-adevăr a fost înscrisă în fișier. Dacă programul se termină pînă la închiderea fișierului, e posibilă situația când o parte din informație ce nu a fost înscrisă pe disc rămîne în zona tampon și în rezultat este pierdută. În afară de aceasta dacă fișierul nu se închide corect, la sfârșitul lui nu va fi înscris în modul necesar codul sfârșit de fișier (*eof*) și următoarea deschidere a fișierului va fi eronată. Deci sistemul operațional va pierde accesul la fișier.

Mai mult ca atît, închiderea fișierului eliberează indicatorul și după aceasta el poate fi folosit pentru accesul la alt fișier sau pentru îndeplinirea altor operații cu fișierul. Spre exemplu fie că trebuie de creat un fișier, de înscris informația în el, apoi de o citit din fișier. Menționăm faptul, că după înscrierea datelor, fișierul trebuie închis și numai după aceasta de-l deschis pentru citire, astfel avînd acces respectiv la datele din fișier:

```
#include<stdio.h>
```



```

void main (void) {
FILE *fişier;
if ((fişier=fopen("info.txt", "w"))==NULL) {
puts ("Fişierul nu poate fi deschis\n");
exit(); }
//Aici vor fi amplasate instrucţiunile de înscriere a informaţiei în fişier.
fclose (fişier);
if ((fişier=fopen("info.txt", "r"))==NULL) {
puts ("Fişierul nu poate fi deschis\n");
exit(); }
//Aici vor fi amplasate instrucţiunile de citire a informaţiei din fişier.
fclose (fişier); }

```

Aici fişierul se deschide în interiorul condiţiei if, adică în timpul deschiderii este controlată valoarea întoarsă de sistemul operaţional la deschiderea fişierului şi dacă este întoarsă valoarea *NULL* se generează mesajul corespunzător şi programul este stopat.

Menţionăm faptul, că unele compilatoare permit înscrierea datelor în fişier prin curăţirea zonei tampon cu ajutorul funcţiei *flush()*. Această funcţie permite, fără a închide fişierul, să înscrie toată informaţia din zona tampon în fişier, apoi această zonă este eliberată de careva date.

12.2. Funcţii de înscriere/citire din fişier.

Limbajul C/C++ conţine mai multe posibilităţi de transmitere a datelor în fişier şi citire din fişier în dependenţă de funcţia folosită:

- Funcţiile *fputc()* şi *putc()* se folosesc în cazul înscrierii unui caracter în fişier sau scoaterii la imprimantă.
- Funcţiile *fgetc()* şi *getc()* se folosesc pentru citirea unui caracter din fişier.
- Pentru a înscrie un şir de caractere în fişier sau a-l imprima la imprimantă este folosită funcţia *fputs()*.
- Pentru a citi un şir de caractere din fişier se foloseşte funcţia *fgets()*.
- Funcţia *fprintf()* este folosită în cazul ieşirii cu format a caracterelor, şirurilor de caractere şi cifrelor pe disc sau la imprimantă.
- Funcţia *fscanf()* se foloseşte pentru citirea cu format a caracterelor, şirurilor de caractere sau cifrelor din fişier.
- Înscrierea unei structuri în fişier e posibilă folosind funcţia *fwrite()*.
- Citirea unei structuri din fişier se efectuează cu ajutorul funcţiei *fread()*.

12.2.1. Înscriere/citire de caractere.

Înscrierea/citirea caracterelor din fișier este forma de bază pentru lucru cu fișiere. Cu toate că nu se bucură de o popularitate mare, aceste operații bine ilustrează principiile de bază în lucrul cu fișierele.

Un caracter poate fi înscris în fișier după, cum a fost spus, cu ajutorul funcției *fputc()* folosind următoarea sintaxă:

fputc(v,fp); unde *v* este o variabilă tip caracterial (*char*), și *fp* – numele indicatorului la fișier.

Următorul exemplu face posibilă înscrierea în fișier a unui set de caractere pînă cînd nu va fi apăsată tasta Enter:

```
#include <stdio.h>
#include <conio.h>
void main (void) {
    FILE *f;
    char lit; clrscr();
    f=fopen("info.txt","w");
    printf ("culege cîteva caractere \n");
    do {
        lit=getch();
        putch (lit);
        fputc(lit,f);} while(lit!='\r');
    fclose (f); }
```

Aici, fișierul este deschis în regimul de acces "w". Și dacă la momentul deschiderii fișierul cu numele "info.txt" nu există, el va fi creat. În ciclul *do* are loc citirea consecutivă a caracterelor de la tastatura cu ajutorul funcției *getch()* și înscrierea lor în fișier cu ajutorul funcției *fputc()*. Este de menționat faptul, că cu același succes aici ar putea fi folosită funcția *putc()* cu aceiași parametri. Ciclul *do* va continua pînă cînd va fi detectată culegerea tastei *ENTER* și după aceasta fișierul este închis.

Cu scopul citirii din fișier a caracterelor sînt folosite funcțiile *getc()* și *fgetc()* care au următoarea sintaxă:

ch_var=getc(fp); unde *ch_var* este variabilă tip catacterial, iar *fp*–indicator la fișier.

Exemplul următor demonstrează cum ar putea fi citită informația din fișierul creat în exemplul precedent:

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

```

void main (void) {
FILE *f1; char lit;
clrscr();
f1=fopen("info.txt", "r");
printf("informația citită:\n");
while( (lit=fgetc(f1))!=EOF)
printf("%c",lit);
fclose(f1); getch(); }

```

Fiind deschis fișierul în regimul "r", este posibilă citirea informației din el. Ciclul *while*, în care are loc citirea consecutivă a caracterelor din fișier este îndeplinit până când nu este detectat simbolul sfârșit de fișier *EOF*, care este înscris la sfârșitul oricărui fișier în momentul închiderii acestuia.

Funcția de citire *fgetc(f1)* folosește alt indicator la același fișier – *f1*. Acest fapt este condiționat de faptul, că chiar dacă la operațiile de înscriere și citire din fișier este folosit același nume de fișier, înscrierea sau citirea este recomandat de efectuat cu ajutorul diferitor indicatori.

12.2.2. Înscriere/citire de șiruri.

În cazul, când e necesar de înscris în fișier seturi de caractere, adică șiruri este binevenită funcția *fputs()*; care are următoarea sintaxă:

fputs (s_var, fp); unde *s_var* este o variabilă tip șir de caractere, iar *fp* – indicator la fișier.

Funcția *fputs()* efectuează înscrierea șirurilor în fișier sau imprimarea lor pe hârtie fără înserarea caracterului *sfârșit de linie*. Pentru ca fiecare șir înscris în așa mod în fișier să înceapă din rând nou, este necesară înserarea manuală a simbolului *sfârșit de linie*.

Următorul exemplu face posibilă înscrierea în fișier a unui set de familii:

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void main (void) {
FILE *k; char fam[15];
clrscr();
printf("culegeți familia\n");
gets (fam);
k=fopen("familia.txt", "w");
while(strlen(fam)>0){

```

```

fputs(fam,k); fputs("\n",k);
printf("culegeți următoarea familie\n");
gets (fam);}
fclose (k);}

```

Aici ciclul *while* va fi repetat pînă cînd va fi introdus un șir de lungime 0. Funcția *fputs()* va înscrie în fișier șirurile fiecare din rînd nou datorită inserării *fputs("\n",k)*; La sfîrșitul prelucrării sale, fișierul *familia.txt* este neapărat închis *fclose(k)*; Este de menționat faptul, că pentru a imprima pe hîrtie familiile culese folosind imprimanta, este necesar de indicat numele fișierului "*prn*" în felul următor: *k=fopen ("prn","w")*; Pentru imprimarea corectă la imprimantă este necesară folosirea șirurilor cu lungime de 81 caractere, cu scopul ca șirul să încapă în întregime în lățimea monitorului, înainte de a fi culeasă tasta ENTER.

Cirirea șirurilor de caractere din fișier este realizată de funcția *fgets()* ce are următoarea sintaxă:

fgets(s_var, l, fp); unde *s_var* este o variabilă tip șir de caractere, *l* – este o variabilă sau constantă de tip întreg, ce indică cantitatea maximă posibilă de caractere în șir, *fp* – este un indicator la fișier.

Următorul exemplu face posibilă citirea familiilor din fișierul "*familia.txt*" creat în exemplul precedent:

```

#include <stdio.h>
#include <conio.h>
void main (void) {
FILE *r; char name[15];
clrscr();
r=fopen("familia.txt", "r");
printf("Informația citită din fișier:\n");
while (fgets(name, 15, r)!=NULL )
printf ("%s",name);
fclose(r); getch(); }

```

Aici ciclul *while* va fi repetat pînă cînd nu va fi defectat codul "sfîrșit de fișier". În cazul citirii informației din fișier la nivel de șiruri de caractere, pentru a indica sfîrșitul fișierului se folosește codul *NULL*, iar codul *EOF* este folosit la citire caracterială. Funcția *fgets()* va citi șirul în întregime pînă la codul „linie nouă”, dacă lungimea lui nu depășește valoarea "*l-1*" indicată în parametrii funcției.

Atrageți atenția la faptul, că funcția *printf("%s",name)*; nu folosește codul "*/n*" pentru deplasarea în rînd nou, fiindcă fiecare șir citit din fișier deacum conține

codul “\n” - linie nouă înscris în fișier în exemplul precedent cu ajutorul funcției *fputs(“\n”,k)*.

12.2.3. Întrare/ieșire cu format.

Funcțiile pentru prelucrarea caracterelor și șirurilor au destinația de înscriere-citire din fișier numai a informației textuale. În caz, când e necesar de înscris în fișier date ce conțin valori numerice este folosită funcția *fprintf()* cu următoarea sintaxă:

fprintf (fp, format, data); unde *fp* este indicatorul la fișierul în care se realizează înscrierea, *format* este șirul de control a formatului datelor înscrise și *data* este lista variabilelor sau valorilor ce trebuie înscrise în fișier. Exemplu:

```
fprinf (f, "%d", cost);
```

Următorul exemplu demonstrează modul de înscriere în fișier a informației despre un set de produse. În fișier vor fi înscrise denumirile, prețurile și cantitățile produselor păstrate într-un depozit:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main (void) {
FILE *f; clrscr();
char name[20], răspuns='y' ;
float pret; int unit;
f=fopen("produs.txt", "w");
while (răspuns=='y' ){
printf ("Culege denumirea produsului\n");
scanf ("%s", name);
printf ("Culege prețul produsului %s\n",name);
scanf ("%f", &pret);
prinf ("Culege cantitatea produsului %s\n", name);
scanf ("%d", &unit);
fprintf (f, "%s %f %d\n", name, pret, unit);
printf ("Doriți să continuați? y/n\n");
răspuns=getch();}
fclose(f);}
```

În rezultatul îndeplinirii acestui program în fișier va fi înscrisă informația despre câteva produse spre exemplu în felul următor:

```
discheta 4.500000 100
mouse 140.000000 3
```

monitor 2000 1

Atrageți atenția la simbolul “\n”- sfârșit la linie la sfârșitul șirului de control a formatului din funcția *fprintf()*, anume datorită lui, informația despre fiecare produs este înscris din rând nou în fișier.

Citirea informației cu format din fișier se realizează cu funcția *fscanf()*, care are aceleași restricții ca și funcția *scanf()*, și folosește următoarea sintaxă:

fscanf(fp, format, data); care este identică cu sintaxa funcției *fprintf()* în ceea ce privește descrierea parametrilor.

În exemplu următor se realizează citirea informației din fișierul “*produs.txt*” despre produsele din depozit înscrise în fișier în exemplu precedent:

```
#include<stdio.h>
#include<conio.h>
void main (void) {
    clrscr();
    FILE *a; char name[20];
    float pret; int unit;
    a=fopen("produs.txt", "r");
    while( fscanf( a, "%s%f%d", name, &pret, &unit)!=EOF) {
        printf("Denumire: %s\n",name);
        printf("pret:      %f\n",pret);
        printf("cantitatea:%d\n",unit); }
    fclose(a); getch(); }
```

Aici citirea datelor din fișier are loc chiar odată cu controlul condiției de continuare a ciclului *while*. Ciclul *while* va fi îndeplinit pînă cînd nu va fi detectat codul de sfîrșit de fișier care în acest caz este *EOF*.

Datele din fișier sînt citite în variabilele *name*, *preț* și *unit* și apoi afișate la monitor.

12.2.4. Fișiere și structuri.

Pentru a înscrie o variabilă tip structură (înscriere) în fișier este folosită funcția *fwrite()*, care are următoarea sintaxă:

fwrite (&struct_var, struct_size, n, fp); unde:

- *&struct_var* este numele variabilei tip structură cu operatorul de adresă, care comunică compilatorului adresa celulei de start din memoria unde este amplasată structura.
- *struct_size* este mărimea structurii. Pentru determinarea mărimii structurii se folosește funcția *sizeof(s)*; unde *s* este numele variabilei de tip structură.

- *n* este un număr întreg, care determină cantitatea de structuri ce vor fi înscrise în fișier dintr-o încercare. Aici este recomandat de folosit valoarea 1. Valori mai mari ca 1 se folosesc în cazul înscrierii în fișier a unui masiv de structuri dintr-o singură încercare.

- *fp* este numele indicatorului la fișier.

Exemplu: *fwrite(&a,size(a),1,f);*

În următorul exemplu este creat un masiv de structuri cu informația despre un grup de studenți ce conține numele studentului, anul nașterii, și balul mediu. Întâi de toate masivul de structuri se completează cu informație, apoi structurile, câte una cu ajutorul ciclului *for* și funcției *fwrite()* sînt înscrise în fișier. Numele fișierului este numit de utilizator și păstrat în variabila "*filename*" de tip șir de caractere:

```
#include<stdio.h>
#include<conio.h>
struct stud {
char nume [15];
int an; float med;};
void main (void) { clrscr();
struct stud x[10]; int i,n;
FILE *f; char filename[12];
float m;
printf("culege numărul de studenț \n");
scanf("%d",&n);
for (i=0; i<n; i++) {
printf ("culege numele studentului\n");
scanf ("%s", x[i]. nume);
printf ("culege anul nașteri \n");
scanf ("%d", x[i]. an);
printf (culege nota medie\n");
scanf ("%f, &m); x[i].med=m; }
printf ("culege numele fișierului \n");
scanf ("%s",filename);
f=fopen(filename, "w");
for(i=0; i<n; i++) fwrite (&x[i], sizeof (x[i]), 1, f);
fclose(f); getch(); }
```

În rezultatul îndeplinirii acestui program, va fi creat un fișier cu numele, care a fost atribuit variabilei *filename* și în el înscrise structurile cu informația despre studenți. Dacă vom deschide fișierul cu ajutorul unui redactor de texte obișnuit vom

observa în el un conținut neînțeles. În realitate informația (sau mai bine zis structurile) ce se află în acest fișier este înțeleasă de compilator și poate fi citită cu ajutorul funcției *fread()* ce are aceeași sintaxă ca și funcția *fwrite()*, în ceea ce privește descrierea parimetrilor:

```
fread (&struct_var, struct_size, n, fp);
```

În următorul exemplu se realizează citirea tuturor înscrierilor (structurilor) din fișierul creat în exemplul precedent și afișarea lor la monitor. Este de menționat momentul că funcția *fread()*, în rezultatul executării sale întoarce o valoare ce corespunde cantității structurilor citite cu succes din fișier. În cazul nostru are loc citirea structurilor câte una din fișier, deci funcția în caz de succes va întoarce valoarea 1. În caz de creare sau de detectare a sfârșitului de fișier funcția va întoarce valoarea 0.

```
#include<stdio.h>
#include<conio.h>
struct stud {
char nume[15];
int an; float med;};
void main (void) {clrscr();
struct stud y [10];
FILE *k; char fn[12]; int i=0;
printf ("culege numele fișierului \n");
scanf ("%s", fn);
k=fopen (fn, "r");
printf ("Informația citită din fișier:\n");
while (fread (&y[i], sizeof(y[i]), 1, k)==1) {
printf ("Numele studentului: %s\n",y[i].nume);
printf ("Anul nașterii: %d\n",y[i].an);
printf ("Media: %f\n",y[i].med); i++;}
fclose(k); getch(); }
```

Următorul tabel conține descrierea tuturor posibilităților de intrare a datelor referitor la fișiere, inclusiv valorile întoarse de funcții în caz de citire eronată:

Tipul de date	Funcții de ieșire	Funcții de intrare	Valoarea întoarsă la citire
<i>caractere</i>	<i>putc(); fputc();</i>	<i>getc(); fgetc();</i>	<i>EOF</i>
<i>șiruri de caractere</i>	<i>fputs();</i>	<i>fgets();</i>	<i>NULL</i>
<i>date cu format</i>	<i>fprintf();</i>	<i>fscanf();</i>	<i>EOF</i>
<i>structuri</i>	<i>fwrite();</i>	<i>fread();</i>	<i>0</i>

Anexa1. Funcții de intrare-ieșire în C/C++

Funcții de ieșire în C

Funcțiile de ieșire în orice limbaj de programare au destinația de a transfera datele din memoria calculatorului la alte dispozitive periferice de ieșire cum ar fi monitorul, imprimanta, un fișier pe disc etc. În timpul procedurii de ieșire se face o copie a datelor care vor fi trimise la dispozitivul de ieșire, păstrându-se originalul lor în memoria calculatorului. În limbajul de programare C există mai multe funcții de ieșire. Și folosirea unei sau altei funcții depinde de tipul datelor ce vor fi extrase și metoda de prezentare a lor. Cea mai simplă sintaxă o au funcțiile de extragere a caracterelor și șirurilor de caractere.

Notă: Toate funcțiile de ieșire din limbajul C sînt susținute și în limbajul C++.

Funcția *puts()*

Funcția *puts()* are destinația de a afișa un șir de caractere la monitor. Sintaxa acestei funcții este următoarea: *puts(P)*; unde P este parametrul funcției și poate fi:

- 1) un literal;
- 2) o constantă de tip șir;
- 3) o variabilă de tip șir.

Definiție: Literal se numește un set concret de caractere care este inclus nemijlocit în instrucțiunile limbajului C/C++ în locul numelui constantei sau a variabilei. În cazul folosirii unui literal în calitate de parametru pentru funcția *puts()* fraza ce trebuie afișată pe ecran este inclusă în parantezele rotunde și delimitată de ghilimele. De exemplu: *puts("Hello world!");* Folosirea unei constante de tip șir în calitate de parametru al funcției *puts()* folosește regulile după cum urmează în exemplu:

```
#define MESAJ "Hello world" void main(void) {puts(MESAJ);}
```

Aici fraza "Hello world" a fost atribuită constantei *MESAJ*, care apoi este folosită ca parametru al funcției *puts()* fără delimitarea cu ghilimele. A treia metodă de folosire a funcției *puts()* cu variabilă de tip șir în calitate de parametru este ilustrată în exemplul următor:

```
void main (void) { char MESAJ[]="Hello world"; puts(MESAJ);}
```

Folosirea constantelor și variabilelor de alt tip în calitate de parametru al funcției *puts()* va duce la eroare de compilare. Unica diferență la folosirea literalelor și constantelor sau variabilelor este faptul că literalul trebuie delimitat de ghilimele iar variabila și constanta se folosesc fara ghilimele. Majoritatea compilatoarelor efectuează trecerea în alt rînd după îndeplinirea funcției *puts()*. Aceasta înseamnă, că după afișarea datelor pe ecran, cursorul în mod automat se deplasează la începutul următorului rînd. Însă această regulă este respectată nu de toate compilatoarele. În

acest caz, pentru trecerea cursorului în alt rind trebuie folosită consecutivitatea de simboluri '\n' destinată afișării pe ecran a simbolului de conducere "trecere în alt rind", în acest caz funcția *puts()* va avea următoarea sintaxă: *puts("Hello world!\n");* Prototipul funcției *puts()* este descris în biblioteca *stdio.h* din această cauză este necesară includerea acestei biblioteci în cadrul programului cu ajutorul directivei *#include<stdio.h>*.

Funcția *putchar()*

Are destinația afișării unui singur simbol la ecran . Ca și în cazul funcției *puts()* în calitate de parametru al funcției poate fi folosit un literal, o constantă sau o variabilă de tip caracter. De exemplu:

literal în calitate de parametru:

```
putchar('C');
```

constantă tip caracter în calitate de parametru:

```
#define lit 'C' void main(void) {putchar(lit);}
```

variabilă tip caracter în calitate de parametru:

```
void main(void) {char lit; lit='C'; putchar(lit);}
```

Menționăm faptul, că cu ajutorul funcției *putchar()* este posibilă afișarea numai a unui singur caracter și instrucția de tipul *putchar('Da');* va duce la eroare de compilare.

Diferența principală la folosirea caracterelor și șirurilor de caractere constă în faptul, că șirurile de caractere sînt delimitate de ghilimele, pe cînd caracterele se delimitate de apostrofe.

Majoritatea compilatoarelor nu efectuează deplasarea cursorului în rind nou după executarea funcției *putchar()* și el rămîne nemijlocit după simbolul afișat. Pentru trecerea în alt rind după afișare se recomandă folosirea simbolului '\n'. Unele compilatoare folosesc pentru afișarea caracterelor și funcția *putch()*, care are sintaxa de folosire identică cu cea a funcției *putchar()*. Prototipul funcției *putchar()* este descris în biblioteca standard *stdio.h*, ceea ce face necesară includerea acestei biblioteci în textul programului cu ajutorul directivei *#include<stdio.h>*. Iar prototipul funcției *putchar()* este descris în biblioteca standard *stdio.h*, ceea ce face necesară includerea acestei biblioteci în textul programului cu ajutorul directivei *#include<stdio.h>*. Iar prototipul funcției *putc()* este descris în biblioteca *conio.h* și folosirea ei va fi însoțită de directiva *#include<conio.h>*.

Funcția *printf()*

Funcțiile *putch()* și *puts()* sînt folosite destul de des, în să posibilitățile lor , cu părere de rău sînt limitate. Aceste funcții nu asigură afișarea unei valori numerice la ecran și pot prelucra numai un singur argument (parametru). În limbajul C există o funcție mult mai universală numită *printf()*. Ea permite afișarea pe ecran a datelor de orice tip și poate prelucra o listă din cîțiva parametri. În afară de aceasta cu ajutorul funcției *printf()* se poate determina formatarea datelor afișate pe display. În cel mai simplu caz funcția *printf()* poate fi folosită în locul funcției *puts()* pentru afișarea unui șir de caractere:

```
#define MESAJ "Hello world!";  
void main(void) { printf(MESAJ); printf("Bine ati venit"); }
```

Ca și în cazul funcției *puts()*, funcția *printf()* poate avea în calitate de parametru un literal, o constantă, o variabilă de tip șir de caractere.

Pentru a afișa pe display mărimi numerice și a avea posibilitate de a formata datele de diferite tipuri, lista de parametri a funcției *printf()* se împarte în două părți:

```
printf("șirul cu format", lista datelor);
```

Primul parametru se numește șir de conducere sau șir cu format. Acest parametru se delimitează cu ghilimele și indică compilatorului în ce poziție din șir trebuie să apară datele. Șirul cu format poate conține orice text împreună cu niște etichete numite indicatori de format care determină tipul datelor și amplasarea lor. Orice indicator de format începe cu simbolul procent (%), după care urmează un caracter ce indică tipul datelor. Așa indicatori sînt: *%d* – număr întreg; *%u* – număr întreg fără semn; *%f* – număr real de tipul float sau double; *%e* – număr real în formă exponențială; *%g* – număr real afișat în format *%f* sau *%e* în dependență de faptul care formă de scriere este cea mai scurtă; *%c* – caracter; *%s* – șir de caractere.

În așa fel prima parte a funcției *printf* se poate scrie în felul următor *printf("%d",...)*; simbolul procent (%) spune compilatorului că după el va urma un indicator de format, iar pantru a afișa pe ecran însăși simbolul procent (%) este necesar de-l scris de 2 ori în felul următor: *printf("%%")*; Litera 'd' indică compilatorului faptul că va trebui afișată o valoare de tip întreg, adică un număr scris în sistemul zecimal.

Partea a doua din lista parametrilor este lista datelor, care poate conține literale, nume de variabile sau constante, valorile căroră este necesar de afișat pe ecran. Lista datelor se desparte de șirul cu forma prin virgulă. Toate elementele din lista datelor deasemenea se despart între ele prin virgulă. Cînd compilatorul prelucrează această funcție, el înlocuiește indicatorii de format cu valorile din lista datelor. De exemplu: *printf("%d",5)*; În timpul îndeplinirii acestei funcții valoarea 5 va fi amplasată în

locul indicatorului de format (%d). Șirul cu format poate conține nu numai indicatori de format, ci și text obișnuit care conține și indicatori de format. De exemplu: *printf("Este ora %d",5);* În rezultatul executării acestui exemplu pe ecran va fi afișat mesajul "Este ora 5". Același effect poate fi primit și la folosirea funcției *puts("Este ora 5");* însă pentru combinarea textului cu valori, constante și variabile numerice e necesară folosirea funcției *printf()*. Exemplu:

```
void main(void) {
    int ora; ora=5; printf("Este ora %d",ora); }
```

Acest exemplu folosește în calitate de parametru o variabilă de tip întreg ora, care este amplasată în lista datelor. Bineînțeles lista datelor poate conține câțiva parametri. De exemplu:

```
void main(void) {
    int ora,min; ora=5; min=25;
    printf("Este ora %d și %d minute",ora,min);}
```

În rezultat vom avea afișat la ecran mesajul: "Este ora 5 și 25 minute". În cazul folosirii câtorva parametri în lista datelor și respectiv în șirul cu format, parametri din lista datelor trebuie să corespundă ca cantitate, poziție și tip cu indicatorii din șirul cu format. În cazul nostru primul simbol %d corespunde variabilei *ora*, iar al doilea – variabilei *min*.

În cadrul unei funcții *printf()* pot fi folosiți nu numai câțiva parametri de același tip, ci și parametri de tipuri diferite. De exemplu:

```
void main(void) {
    int cant; float pret; pret=5.21; cant=3;
    printf("Prețul este de %f lei pentru %d kg.",pret,cant);}
```

În rezultat va fi afișat mesajul: "Prețul este de 5.210000 lei pentru 3 kg." Aici întradevăr parametrii din lista datelor corespund ca cantitate, poziție și tip cu indicatorii de format din șirul cu format.

Funcția *printf()* nu trece automat cursorul în alt rind după afișarea datelor. Adică după afișare cursorul rămâne în același rind nemijlocit după ultimul caracter afișat. Pentru trecerea cursorului în alt rind aici va fi folosită consecutivitatea de conducere '\n', care trece cursorul pe prima poziție a rîndului următor ca și în cazul funcției *puts()*. De exemplu: *printf("Prețul este de %f lei \n pentru %d kg.",5.21,3);* Aici după afișare mesajul va fi amplasat în 2 rînduri:

```
Prețul este de 5.210000 lei
pentru 3 kg.
```

La fel ca și simbolul '\n' aici pot fi folosite și simbolurile: '\a', '\b', '\t', '\v', '\r' ș.a.

Funcția *printf()* poate fi folosită pentru dirijarea cu formatul datelor. Aici, determinarea cantității de poziții pe care va fi afișat numărul se determină cu indicatorul de lățime a câmpului.

Fără folosirea indicatorului de lățime a câmpului cifrele vor fi afișate în formatul standard pentru tipul de date corespunzător. De exemplu cifrele reale vor fi afișate cu 6 poziții după virgulă. În exemplul precedent prețul 5.210000 este afișat întradevăr cu 6 poziții după virgulă, acest format se poate de schimbat după dorința utilizatorului. Folosind indicatorul de lățime a câmpului se poate determina cantitatea de poziții pe care va fi afișată o valoare de orice tip de date. De exemplu:

```
printf("Prețul este de %.2f lei \n pentru %d kg.", 5.21, 3);
```

Aici numărul real va fi afișat cu 2 poziții după virgulă: 5.21. Iar în versiunea *printf("Prețul este de %6.2f lei \n pentru %d kg.", 5.21, 3);* vom avea `__5.21` unde `'_'` este spațiu. În general indicatorul de lățime a câmpului pentru numerele reale are următoarea formă `%k.rf`, unde *k* este numărul total de poziții (inclusive virgula) pe care va fi afișată cifra reală, *r* este numărul de poziții după virgulă. Dacă *k* va fi mai mare decât *r*+2 atunci vor fi inserate spații înaintea numărului în cantitate de *k-r*+2. Sub cifra 2 aici se are în vedere 2 poziții din numărul real: o poziție de la cifra întreagă pînă la virgulă și o poziție este însăși virgula. Dacă *k* va fi mai mic ca *r*+2, indicatorul de lățime a câmpului pur și simplu va fi ignorat. Indicatorul de lățime a câmpului de forma `%6.4f` va prezenta cifra 5.21 în formă de 5.2100. Pentru numerele întregi, caractere și șiruri de caractere indicatorul de lățime a câmpului are forma `%kd`, `%kc`, `%ks`, unde *k* este numărul total de poziții pe care va fi afișată valoarea. În același timp dacă *k* va fi mai mare ca lungimea reală a valorii afișate, înaintea acestei valori vor fi inserate spații în cantitate de *k-p*, unde *p* este numărul de poziții pe care real este amplasată valoarea afișată. Iar dacă *k* va fi mai mic ca lungimea reală a valorii, indicatorul de lățime a câmpului pur și simplu va fi ignorat. De exemplu:

```
printf("Este ora %3d", 5); //în rezultat "Este ora __5";
```

```
printf("Mă numesc %2s", Anatol); //în rezultat "Mă numesc Anatol";
```

```
printf("Litera %2c", 'A'); //în rezultat "Litera _A";
```

unde simbolul `'_'` reprezintă spațiu.

Afișarea informației în C++

Toate funcțiile de ieșire analizate mai sus sînt valabile atît pentru limbajul C, cît și pentru C++. Însă limbajul C++ are o posibilitate adăugătoare de afișare a datelor de orice tip. Limbajul C++ conține fluxul standard de ieșire `"cout"`, care permite împreună cu operatorul inserării, compus din două simboluri (`<<`) mai mic, afișarea literalelor, valorilor constantelor și variabilelor fără folosirea indicatorilor de format.

Pentru folosirea fluxului *cout* este necesar de inclus în textul programului C++ fișierul `<iostream.h>` cu ajutorul directivei `#include<iostream.h>`, din cauza că el conține descrierea fluxurilor de intrare și ieșire în C++. Structura instrucțiunii ce folosește fluxul *cout* este următoarea: `"cout<<lista_de_date;"` unde simbolul `'<<'` este operatorul inserării și indică compilatorului necesitatea afișării listei de date ce urmează după el. În calitate de informația afișată cu ajutorul fluxului *cout* pot servi literalii, nume de constante și variabile de orice tip. Mai mult ca atât, folosind unul și același flux de ieșire se poate afișa pe ecran câțiva argumenti, cu condiția că ei vor fi delimitați între ei prin operatorul inserării. De exemplu:

```
#include<conio.h>
#include<iostream.h>
void main(void) { clrscr();
const cant=3;
int ora, min; float suma;
suma=5.21; ora=9; min=20;
cout<<"Hello world\n";
cout<<"Este ora "<<9;
cout<<"\nEste ora "<<ora<<" si "<<min<<" minute\n";
cout<<"Pretul este de "<<suma<<" lei pentru "<<cant<<" kg";
getch();}
```

Deasemenea ca și funcția *printf()* fluxul *cout* nu trece cursorul în rind nou automat după afișarea datelor. Pentru prezentarea comodă a datelor aici este necesară folosirea consecutivității de conducere `'\n'` după cum este arătat în exemplul precedent.

Funcții de intrare în C.

Procesul de intrare a datelor presupune introducerea informației necesare pentru lucrul normal al programului. Informația introdusă se amplasează în variabile, aceasta înseamnă că valorile venite de la utilizator în urma afișării comentariului corespunzător pentru introducere sînt atribuite ca valori variabilelor ce se păstrează în memorie. În general intrarea datelor poate fi efectuată de la diferite surse așa ca tastatura, memoria, unitatea de disc rigid sau flexibil, scanner, CD-ROM ș.a.

Intrarea datelor este un process ce determină lucrul de mai departe al programului. Corectitudinea datelor de intrare influențează direct asupra corectitudinii rezultatelor obținute cu folosirea acestor date de intrare. Datele de intrare pot fi atribuite ca valori numai variabilelor și nici într-un caz constantelor, valorile cărora nu se schimbă pe întreg parcursul de îndeplinire a programului. Dacă variabila în care va fi amplasată o

valoare de la un dispozitiv de intrare deja are o valoare oarecare, atunci valoarea nou venită va înlocui valoarea veche. În continuare vor fi studiate și analizate numai posibilitățile de intrare de la tastatură, lăsând pentru temele viitoare studierea mecanismelor intrării datelor din alte surse.

Funcția *getchar()*

Funcția *getchar()* face posibilă introducerea de la tastatură a unui singur simbol. Majoritatea compilatoarelor nu fac diferență între valori de tipul *int* și *char* la folosirea funcției *getchar()*, fapt condiționat de standardul K&R C. Dacă variabila va fi declarată de tip *int*, funcția *getchar()* va primi-o în calitate de valoare de intrare și compilatorul nu va semnala eroare, însă valoarea caracterului culeasă de la tastatură va fi conversată într-un număr întreg egal cu codul corespunzător acestui caracter din tabelul cu caractere ASCII.

Sintaxa de folosire a acestei funcții este următoarea: *var=getchar();* , unde *var* este numele variabilei cărei îi va fi atribuit caracterul cules de la tastatură.

Aici funcția *getchar()* este chemată folosind altă sintaxă în comparație cu funcțiile *puts()*, *putch()*, *printf()*. Inscripția folosită înseamnă: de a atribui variabilei cu numele *var* valoarea primită în rezultatul executării funcției *getchar()*. Funcția *getchar()* nu folosește argument, din această cauză parantezele rotunde de după numele funcției rămân goale. După ce utilizatorul culege o tastă de la tastatură, *getchar()* afișază simbolul introdus pe ecran. În acest caz nu e necesară culegerea tastei *Enter* după îndeplinirea funcției, din cauza că *getchar()* face posibilă introducerea numai unui singur simbol, după ce programul trece la îndeplinirea de mai departe. Valoarea introdusă este atribuită variabilei îndată după ce a fost cules un caracter. În timpul folosirii funcției *getchar()*, în textul programului trebuie inclus fișierul *<stdio.h>*, care conține descrierea prototipului acestei funcții.

Unele compilatoare C și C++ folosesc funcția *getch()* analogică cu funcția *getchar()*. Însă descrierea prototipului funcției *getch()* se află în fișierul *<conio.h>*, care trebuie inclus în textul programului: *#include<conio.h>*. Exemplu:

```
#include<stdio.h>
#include<conio.h>
void main(void) {char lit, lit1; lit=getchar(); lit1=getch(); putchar(lit);
putchar('\n'); putchar(lit1);}
```

În timpul îndeplinirii programului compilatorul nu are indicații implicite de a opri îndeplinirea acestuia după executarea întregului program. Acest fapt este necesar pentru a fi posibilă vizualizarea și analiza rezultatelor nemijlocit după sfârșitul

executării programului. Funcțiile *getch()* și *getchar()* ne dau posibilitatea de a obține acest efect folosind următoarea sintaxă: *getch()* sau *getchar()*. Exemplu:

```
#include<stdio.h>
#include<conio.h>
void main(void) {char lit, lit1; lit=getchar(); lit1=getch();
putchar(lit); putchar('\n'); putchar(lit1);
getch();}
```

În acest caz, ajungând la sfârșitul programului și întâlnind funcția *getch()*, compilatorul va opri executarea programului până când va fi culeasă tasta *Enter*.

Notă: Limbajul C/C++ conține funcția *gets()* destinată introducerii de la tastatură a șirurilor de caractere. Sintaxa și principiul de lucru al acestei funcții vor fi studiate la tema “Șiruri de caractere”.

Funcția *scanf()*

Funcția de intrare *scanf()* face posibilă introducerea în calculator a datelor de orice tip. Funcția scanează tastatura, determină tastele care au fost culese și apoi interpretează informația culeasă cu ajutorul indicatorilor de format, ce se află în componența funcției. Ca și în cazul funcției *printf()* funcția *scanf()* poate avea mai mulți argumenti făcând posibilă introducerea valorilor de tip întreg, caracterial și șir de caractere în același timp, adică în cadrul unei funcții. Lista de parametri ai funcției *scanf()* este compus din două părți ca și la funcția *printf()*. Prima parte este șirul cu format, iar a doua lista cu date. Șirul cu format conține indicatori de format numiți convertori de simboluri, care determină modul în care trebuie să fie interpretate datele de intrare. Lista datelor conține lista variabilelor în care vor fi păstrate valorile introduse. Sintaxa funcției *scanf()* este următoarea:

```
scanf(“șirul cu format”, lista datelor);
```

De exemplu:

```
#include<stdio.h>
void main(void) { int a; float b;
printf(“Culege valorile a întreg și b real\n”);
scanf(“%d%f”, &a, &b);
printf(“a=%d b=%f\n”, a, b); getch(); }
```

Să analizăm funcția *scanf(“%d%f”, &a, &b);*

Folosind funcția *scanf()* în timpul când se introduc datele, este necesar în lista de date de indicat adresa variabilei căreia îi va fi atribuită valoarea curentă. Adresa de memorie rezervată variabilei în timpul declarării sale se poate afla folosind operatorul de adresă ‘&’. În timpul când funcția *scanf()* întâlnește formatul variabile a, ea îl

determină ca format al unui număr întreg și scanează tastatură așteptând culegerea unui număr întreg pe care apoi îl înscrie pe adresa de memorie rezervată variabilei *a* ocupând doi octeți. Odată fiind înscris în celula de memorie rezervat variabilei *a*, numărul corespunzător devine valoare a acestei variabile. La fel se întâmplă și în cazul cu variabila de tip real *b*. După prelucrarea tuturor variabilelor din lista parametrilor, funcția *scanf()* își termină lucrul în cazul dacă a fost culeasă tasta *Enter*.

Procedura de intrare în C++

Compilatoarele limbajului C++ susțin funcțiile de intrare *gets()*, *getchar()* și *scanf()* despre care s-a vorbit mai sus. În afară de aceste funcții C++ mai conține o posibilitate de intrare a datelor. Fluxul standard de intrare *cin* în ansamblu cu două simboluri (*>>*) “mai mare” ce se numesc operatori de extragere, face posibilă intrarea datelor de orice tip de la tastatură și are următoarea sintaxă: *cin>>var;* unde *var* este numele variabilei cărei îi va fi atribuită valoarea citită de la tastatură. De exemplu:

```
#include<stdio.h>
#include<iostream.h>
void main(void) { int a; float b;
printf(“Culege valorile a întreg și b real\n”);
cin>>a>>b;
cout<<”a=”<<a<<” b=”<<b;    getchar(); }
```

În acest exemplu cu ajutorul fluxului standard de intrare *cin* au fost scanate de la tastatură și atribuite valori variabilelor *a* și *b*. În acest caz nu e necesar de indicat adresa de memorie unde va fi amplasată valoarea scanată ca în cazul funcției *scanf()*, ci se indică numai numele variabilei. În afară de aceasta fluxul standard de intrare are posibilitatea de a determina automat tipul valorii introduse și nu e necesară folosirea indicatorilor de format. Iar în caz de lucru cu mai multe variabile, ele trebuie despărțite cu ajutorul operatorului de extragere.

Anexa 2. Funcții matematice.

După cum a mai fost spus stilul de programare în C se caracterizează prin tendința de a evidenția un număr mare de funcții nu prea voluminoase, astfel, ca prelucrarea datelor în aceste funcții să nu depindă de celelalte părți ale programului. Acest lucru face programul destul de înțeles și dă posibilitatea de a introduce ușor corecții în unele funcții fără a tangenta celelalte. În marea majoritate acest fapt este cert în cazul funcțiilor create de utilizator pentru a împărți problema în câteva sarcini mai mici dar mai simple. Însă limbajul C conține și o mulțime de funcții standarde care ușurează cu mult programarea. Aceste funcții oferă o varietate de facilități. În

plus, fiecare programator își poate construi propria sa bibliotecă de funcții care să înlocuiască sau să extindă colecția de funcții standard ale limbajului.

Ca atare limbajul de programare C conține în arsenalul său de instrucțiuni foarte puține instrumente pentru a putea fi numit limbaj ce se află pe primele locuri în top-ul limbajelor de programare de nivel înalt. Renumita capacitate de a prelucra expresii matematice complicate în limbajul C este datorată bibliotecilor de funcții standarde, care ușurează cu mult rezolvarea unor situații destul de complicate. Unele din așa funcții matematice ale limbajului C întâlnite mai des în practica de programare vor fi studiate în prezentul paragraf. Menționăm faptul, că aceste funcții sînt valabile și în limbajul C++ și folosirea lor într-un program în C sau C++ necesită includerea fișierului `mat.h` cu ajutorul directivei `#include<math.h>`.

1. Funcția `abs(x)`.

Prototip: `int abs(int x); double fabs(double x); long int labs(long int x);`

Efect: Întoarce valoarea absolută a numărului `x`.

Exemplu:

```
#include<math.h>
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
int x,y;
cout<<"Culegeti valoarea x\n";
cin>>x; y=abs(x);
cout<<"modulul lui "<<x<<" este = "<<y;
getch(); }
```

2. Funcția `cos(x)`.

Prototip: `double cos(double x); long double cosl(long double x);`

Efect: Întoarce valoarea cosinus a numărului `x` [`cos(x)`];

Exemplu:

```
#include<math.h>
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double x,y;
cout<<"Culegeti valoarea x\n";
cin>>x; y=cos(x);
cout<<"cosinusul lui "<<x<<" este = "<<y;
getch(); }
```

3. Funcția $\sin(x)$.

Prototip: `double sin(double x); long double sinl(long double x);`

Efect: Întoarce valoarea sinus a numărului x [$\sin(x)$];

Exemplu: vezi funcția $\cos(x)$.

4. Funcția $\tan(x)$.

Prototip: `double tan(double x); long double tanl(long double x);`

Efect: Întoarce valoarea tangentei a numărului x [$\tan(x)$];

Exemplu: vezi funcția $\cos(x)$.

5. Funcția $\arccos(x)$.

Prototip: `double acos(double x); long double acosl(long double x);`

Efect: Întoarce valoarea arccosinus a numărului x [$\arccos(x)$];

Exemplu:

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

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

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

```
void main(void) {clrscr();
```

```
double x,y;
```

```
cout<<"Culegeti valoarea x\n";
```

```
cin>>x; y=acos(x);
```

```
cout<<"arccosinusul lui "<<x<<" este = "<<y;
```

```
getch(); }
```

6. Funcția $\arcsin(x)$.

Prototip: `double asin(double x); long double asinl(long double x);`

Efect: Întoarce valoarea arcsinus a numărului x [$\arcsin(x)$];

Exemplu: vezi funcția $\arccos(x)$.

7. Funcția $\arctan(x)$.

Prototip: `double atan(double x); long double atanl(long double x);`

Efect: Întoarce valoarea arctangentei a numărului x [$\arctg(x)$];

Exemplu: vezi funcția $\arccos(x)$.

8. Funcția $\log(x)$.

Prototip: `double log(double x); long double logl(long double x);`

Efect: Întoarce logaritmul natural al numărului x ;

Exemplu:

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

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

```
void main(void) {clrscr();
```

```
double x,y;
```

```
cout<<"Culegeti valoarea x\n";
cin>>x; y=log(x);
cout<<"logaritmul natural a lui "<<x<<" este = "<<y;
getch(); }
```

9. Funcția log10(x).

Prototip: double log10(double x); long double log10l(long double x);

Efect: Întoarce logaritmul zecimal al numărului x;

Exemplu: Vezi funcția log(x);

10. Funcția exp(x).

Prototip: double exp(double x); long double expl(long double x);

Efect: Întoarce valoarea e^x , unde $e=2.7$ este constantă.

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double x,y;
cout<<"Culegeti valoarea x\n";
cin>>x; y=exp(x);
cout<<"exponenta lui "<<x<<" este = "<<y;
getch(); }
```

11. Funcția ldexp(a,b);

Prototip: double ldexp(double a, int b); long double ldexpl(long double a, int b);

Efect: Întoarce valoarea $2^b * a$;

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double a,y;
cout<<"Culegeti valoarea a\n";
cin>>a; y=ldexp(a,3);
cout<<"2 la puterea 3 înmulțit cu "<<a<<" este = "<<y;
getch(); }
```

12. Funcția frexp(x,y).

Prototip: double frexp(double x, int *y);

long double frexpl(long double x, int *y);

Efect: Întoarce valoarea $x * 2^y$ calculând deasemenea și valoarea lui y.

Exemplu: În calitate de necunoscută aici este folosită numai variabila x. Având un x=8, operația $k = \text{frexp}(x, y)$ calculează cifra reală (k), care trebuie înmulțită la 2^y pentru a primi rezultatul egal cu x=8, determinându-l în același timp și pe y (valoarea puterii la care va trebui ridicată cifra 2). În cazul nostru x=8 și $k = \text{frexp}(x, y)$; va avea următoarele rezultate: y=4; k=0.5; adică $0.5 = \frac{8}{2^4}$

```
#include <conio.h>
#include <math.h>
#include <stdio.h>
int main(void){clrscr();
    double k,x;    int y;
    x = 8.0;
    k = frexp(x,&y);
    printf("numarul %f va fi primit in rezultatul\n",x);
    printf("inmultirii lui 2 la puterea %d si %f ",y,k);
    getch(); }
```

13. Funcția pow(x,y).

Prototip: double pow(double x, double y); long double powl(long double x, long double y);

Efect: Întoarce valoarea x^y ;

Exemplu:

```
#include <iostream.h>
#include <conio.h>
void main(void) {clrscr();
    double x,y,k;
    cout<<"Culegeti valoarea x\n";
    cin>>x;
    cout<<"Culegeti valoarea y\n";
    cin>>y;
    k=pow(x,y);
    cout<<x<<" la puterea "<<y<<" este = "<<k;
    getch(); }
```

14. Funcția pow10(x).

Prototip: double pow10(int x); long double pow10l(int x);

Efect: Întoarce valoarea 10^x ;

Exemplu:

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

```
void main(void) {clrscr();
double y; int x
cout<<"Culegeti valoarea x\n";
cin>>x; y=pow10(x);
cout<<"10 la puterea "<<x<<" este = "<<y;
getch(); }
```

15. Funcția sqrt(x).

Prototip: double sqrt(double x);

Efect: Întoarce valoarea \sqrt{x} ;

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double y,x;
cout<<"Culegeti valoarea x\n";
cin>>x; y=sqrt(x);
cout<<"rădăcina patrată din "<<x<<" este = "<<y;
getch(); }
```

16. Funcția ceil(x).

Prototip: double ceil(double x); long double ceill(long double x);

Efect: Întoarce valoarea ROUND UP a numărului x. Adică îl rotungește pe x în partea de sus.

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double y,x;
cout<<"Culegeti valoarea x\n";
cin>>x; y=ceil(x);
cout<<"valoarea rotunjită a lui "<<x<<" este = "<<y;
getch(); }
```

17. Funcția floor(x).

Prototip: double floor(double x); long double floorl(long double x);

Efect: Întoarce valoarea ROUND DOWN a numărului x. Adică îl rotungește pe x în partea de jos.

Exemplu: vezi funcția ceil(x).

18. Funcția fmod(x,y).

Prototip: double fmod(double x, double y); long double fmodl(long double x, long double y);

Efect: Întoarce valoarea restului de la împărțirea lui x la y.

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double y,x,k;
cout<<"Culegeti valoarea x\n";
cin>>x;
cout<<"Culegeti valoarea y\n";
cin>>y;
k=fmod(x,y);
cout<<"restul de la împărțirea "<<x<<" la "<<y<<" este = "<<k;
getch(); }
```

19. Funcția modf(x,y).

Prototip: double modf(double x, double *y); long double modfl(long double x, long double *y);

Efect: Întoarce partea fracționară a numărului real x, păstrându-se în y și valoarea părții întregi a numărului real x.

Exemplu:

```
#include<iostream.h>
#include<conio.h>
void main(void) {clrscr();
double y,x,k;
cout<<"Culegeti valoarea x\n";
cin>>x;
k=modf(x,&y);
cout<<"Cifra "<<x<<" are partea întreagă = "<<y<<
"și partea fracționară = "<<k; getch(); }
```

20. Funcția div(x,y).

Prototip: div_t div(int x, int y); ldiv_t ldiv(long int x, long int y);

unde tipul div_t este declarat în felul următor:

```
typedef struct { long int quot; /* partea întreagă
long int rem; /* restul
}div_t;
```

Notă: Este necesară includerea bibliotecii <stdlib.h>;

Efect: Întoarce o valoare compusă de tip structură `div_t`, care conține 2 valori: partea întreagă și restul de la împărțirea lui `x` la `y`.

Exemplu:

```
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>
div_t a;
void main (void) { clrscr();
a=div(16,3);
printf("16 div 3 =%d, restul=%d\n",a.quot,a.rem);
getch();}
```

21. Funcția `randomize()`. Inițializator de generator al numerelor aleatoare.

Prototip: `void randomize(void);`

Efect: inițializează generatorul de numere aleatoare.

Notă: Este necesară includerea bibliotecii <stdlib.h>;

Exemplu: Se folosește împreună cu funcția `random(x)`. Vezi funcția `random(x)`;

22. Funcția `random(x)`. Generator de numere aleatoare.

Prototip: `int random(int x);`

Efect: Întoarce o valoare întreagă aleatoare în intervalul de la 0 la $(x-1)$;

Notă: Este necesară includerea bibliotecii <stdlib.h>;

Exemplu:

```
#include <stdlib.h>
#include <stdio.h>
#include<conio.h>
void main(void)
{int i; clrscr();
randomize();
for(i=50;i<60;i++)
printf("cifra aleatoare: %d\n",random(i));
getch();}
```

23. Funcția `rand()`. Generator de numere aleatoare.

Prototip: `int rand(void)`

Efect: Întoarce o valoare întreagă aleatoare în intervalul de la 0 la `RAND_MAX`, unde `RAND_MAX` depinde de realizarea limbajului.

Notă: Este necesară includerea bibliotecii <stdlib.h>. Nu necesită inițializare.

Exemplu:


```

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void main(void)
{int i; clrscr();
for(i=0;i<15;i++)
printf("cifra aleatoare: %d\n",rand());
getch(); }

```

Anexa 3. Funcții folosite la prelucrarea șirurilor de caractere.

Majoritatea compilatoarelor C/C++ au incorporate funcții speciale pentru lucrul cu șirurile. Evident, că pentru așa scopuri se poate crea funcții proprii, dar este mai efectiv de folosit funcțiile din bibliotecile standarde. Prototipurile acestor funcții sînt descrise în biblioteca *string.h* și deci pentru folosirea lor, în program trebuie inclusă această bibliotecă folosind sintaxa: *#include<string.h>*. Unele din aceste funcții sînt:

1. Funcția *strcat()*;

Prototip: *char *strcat (char *dest, const char *sursa);*

Efect: Adaugă șirul sursă la sfîrșitul șirului destinație

Exemplu:

```

#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void) {
char dest [50]; char sursa [5];
dess= "Turbo "; sursa = "C++ ";
strcat (dest,sursa); puts(dest); getch(); }

```

Aici rezultatul va fi: *dest=Turbo C++*.

2. Funcția *strcmp()*;

Prototip: *int strcmp (const char * S1, const char* S2);*

Efect: Compară două șiruri.

Limbajul C nu permite compararea a două șiruri în forma: *if (S1==S2) ;*

Aici compararea este făcută cu ajutorul funcției *strcmp()* care întoarce valoare nulă, în caz că șirurile sînt identice, sau o valoare diferită de zero, în caz că șirurile nu coincid. După executarea funcției *strcmp()*, va fi întoarsă o valoare întreagă care va fi: mai mică ca 0 dacă *S1<S2*;
mai mare ca 0 dacă *S1>S2*;

egală cu 0 dacă $S1==S2$;

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void) {
    char S1 [30]; char S2 [30]; int k;
    gets (S1); gets (S2); k=strcmp(S1,S2);
    If (k==0) puts ("sirurile coincid"); else puts ("sirurile nu coincid");
    getch();}
```

3. Funcția strcmpi();

Prototip: int strcmpi (const char *S1, const char *S2);

Efect: compară 2 șiruri fără a lua în considerație registrele simbolurilor.

Exemplu: E identică cu funcția strcmp().

4. Funcția strncmp();

Prototip: int strncmp(const char *S1, const char *S2, size_t k);

Efect: funcția *strncmp()* compară un număr dat de caractere în 2 variabile de tip șir de caractere.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
    char a [40]="Turbo", b[40]="Pascal "; int k;
    k=strncmp(a,b,1); printf("%d",k); }
```

aici va fi comparat primul caracter din șirul a cu primul caracter din șirul b.

După executarea funcției *strncmp()*, va fi întoarsă o valoare întreagă care va fi:

mai mică ca 0 dacă $a < b$;

mai mare ca 0 dacă $a > b$;

egală cu 0 dacă $a = b$;

Aici va fi primit raspuns $k > 0$;

5. Funcția strncmpi();

Prototip: int strncmpi(const char *S1, const char *S2, size_t k);

Efect: Compară un număr dat de caractere, începînd cu primul, din 2 șiruri de caractere fără a face diferență între caractere minuscule și cele majuscule.

Exemplu: Echivalent cu funcția *strncmp()*.

6. Funcția `strlen()`;

Prototip: `size_t strlen(const char *S);`

Efect: Determină lungimea șirului de caractere S.

În majoritatea cazurilor lungimea șirului nu coincide cu lungimea masivului în care se află șirul. Adică lungimea masivului este mai mare. Funcția `strlen()` determină lungimea reală a șirului și întoarce o valoare de tip întreg egală cu cantitatea de caractere ce se află în șir.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void) {
    char name[20]; int k;
    puts ("culege numele"); gets (name);
    k=strlen(name); printf ("Numele dvs are %d caractere",K); getch(); }
```

7. Funcția `strcpy()`;

Prototip: `char *strcpy(char *S1, const char *S2);`

Efect: Copie șirul S2 în șirul S1. După îndeplinirea funcției `strcpy (S1,S2)`; șirul S1 își va pierde valoarea inițială și va avea valoarea nouă din S2. Iar S2 va rămîne neschimbat.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void) {
    char name [30]; char fam [30];
    puts ("Culege numele"); gets (nume);
    puts ("culege familia"); gets (fam);
    strcpy (nume, fam);
    puts(nume); puts(fam); getch(); }
```

În rezultatul îndeplinirii acestui exemplu numele va fi identic cu familia.

8. Funcția `strcspn()`;

Prototip: `size_t strcspn (const char *S1, const char *S2);`

Efect: Determină poziția caracterului din șirul S1, care primul a fost întâlnit în șirul S2. Întoarce o valoare de tip întreg egală cu numărul de ordine a acestui caracter.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char name[20],fam[20]; int k;
puts ("culege numele"); gets (name);
puts ("culege familia"); gets (fam);
k=strcspn(name,fam);
printf ("Simbolul %c din %s primul a fost gasit in %s",
name[k],name,fam);
getch(); }
```

Dacă în cazul nostru name="Stepan" și fam="Ivanov", atunci rezultatul funcției `k=strcspn(name,fam);` va fi `k=4`, din cauză că pe locul patru în numele "Stepan" se află caracterul "a" care primul a fost depistat în "Ivanov".

9. Funcția strspn();

Prototip: `size_t strspn(const char *S1, const char *S2);`

Efect: Determină poziția caracterului din șirul S1 începând cu care S1 diferă de S2. Întoarce o valoare tip întreg egală cu poziția acestui caracter.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char name[20],fam[20]; int k;
puts ("culege numele"); gets (name);
puts ("culege familia"); gets (fam);
k=strspn(name,fam);
printf ("Începând cu simbolul %c șirul %s diferă de %s",
name[k],name,fam);
getch(); }
```

În caz că name="Stepan", iar fam="Stoianov" lui k va fi atribuită valoarea 2 din cauză că anume începând cu caracterul cu numărul de ordine 2 variabila *nume* diferă de variabila *fam*.

10.Funcția strdup();

Prototip: `char *strdup (const char *S);`

Efect: Dublează șirul de caractere S. În caz de succes funcția *strdup()* întoarce ca valoare indicatorul adresei de memorie, ce conține șirul dublat. Și întoarce valoare nulă în caz de eroare. Funcția *strdup(S)* face o copie a șirului S, obținând spațiu prin apelul funcției *malloc()*. După folosirea șirului dublat programatorul trebuie să elibereze memoria alocată pentru el.

Exemplu :

```
#include <alloc.h>
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a []="UTM"; char *b;
b=strupr(a); puts (b); free (b);}
```

Aici *a* este un șir de caractere, iar *b* un indicator la adresa de memorie unde se va înscrie șirul dublat. La executarea funcției *strupr()* valoarea șirului din *a* este copiat într-o locație de memorie, adresa căreia se află în *b*. Funcția *puts()* afișază conținutul șirului dublat la monitor. Funcția *free()* eliberează memoria ocupată până acum de șirul dublat.

11.Funcția *strlwr()*;

Prototip: *char *strlwr (char *S);*

Efect: Trece toate caracterele din șirul S în echivalentul lor minuscul. În calitate de parametru funcția folosește o variabilă de tip șir de caractere. În rezultatul executării acestei funcții, dacă în șir se vor conține caractere majuscule, ele vor fi transformate în echivalentul lor minuscul, iar dacă în șir caractere majuscule nu se vor conține - șirul va rămâne neschimbat.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[10]= "Pascal";
strlwr(a);
puts (a); getch(); }
```

12.Funcția *strupr()*;

Prototip: *char *strupr(char *S);*

Efect: transformă toate caracterele din șir în echivalentul lui majuscul.

Exemplu: Echivalent cu funcția *strlwr()*;

13. Funcția strncat()

Prototip: `char *strncat (char *dest, const char *sursa, size_t k);`

Efect: funcția `strncat()` adaugă un număr egal cu k caractere de la începutul șirului `sursa` la sfârșitul șirului `dest`.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
    char a[20]="Turbo"; b[10]="Pascal";
    strncat (a, b, 3); puts (a); }
```

În rezultatul executării acestui exemplu primele 3 caractere din șirul `b[10]`, (adica subșirul "Pas") vor fi adăugate la sfârșitul șirului `a[20]`. Rezultatul executării funcției `strncat()` va fi de tip șir de caractere și se va conține în variabila `a`. După executarea exemplului variabila `a` va conține valoarea "TurboPas".

14. Funcția strncpy();

Prototip: `char *strncpy (char *dest, const char *sursa, size_t n);`

Efect: Copie un număr dat de caractere dintr-un șir în altul;

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
    char a[40]="turbo"; b [40]="basic ";
    strncpy (a,b,2); puts (a); }
```

Funcția `strncpy()` înscrie un număr N dat de caractere din șirul sursă la începutul șirului destinație. Dacă șirul destinație va avea lungime mai mare ca N , atunci rezultatul va avea începutul egal cu caracterele copiate, iar sfârșitul inițial. Valoarea rezultatului se va conține în șirul destinație. În exemplu de mai sus variabila `a` va avea valoare egală cu "barbo".

15. Funcția strnset();

Prototip: `char *strnset (char* s, int ch, size_t n) ;`

Efect: Funcția `strnset()` copie caracterul `ch` pe primele n locuri din șirul `*S`. În caz că $n > \text{strlen}(s)$, atunci n va deveni egal cu `strlen(s)`

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[15] = "student" , b=W ;
strnset (a,b,3); puts (a) ;}
```

În urma executării funcției strnset() va fi întoarsă o valoare de tip șir care va fi înscrisă în șirul destinație . Din exemplu de mai sus $a = "WWWdent"$.

16.Funcția strrev();

Prototip char *strrev(char *s);

Efect: Funcția strrev() inversează șirul de caractere S. După execuția funcției strrev() primul caracter din șir va fi schimbat cu locul cu ultimul caracter, caracterul 2 cu penultimul, etc. fără a lua în considerație caracterul nul.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char s1[10]="student";
printf("Șirul inițial - %s\n",s1);
strrev(s1);
printf("Șirul final - %s\n",s1);}
```

După executarea programului va fi primit mesajul: Șirul final – tneduts.

17.Funcția strstr ();

Prototip: char *strstr(const char *s1, const char *s2);

Efect: Funcția strstr() determină dacă șirul S2 se conține în șirul S1. Funcția întoarce un indicator la caracterul din S1 începînd cu care a fost depistat șirul S2.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char S1[20], S2[20], rez;
S1 = "international"; S2 = "nation";
rez = strstr (S1,S2); printf ("subsirul este : %s",rez); }
```

Rezultatul: “subșirul este: national ;”. Dacă șirul S2 n-a fost depistat în S1, funcția strstr() întoarce valoarea “null”.

18.Funcția strchr();

Prototip: char *strchr(const char *S, int c);

Efect: Scanează șirul S în căutarea caracterului c. În caz de succes funcția întoarce un indicator la caracterul din S care primul a fost găsit identic cu caracterul c. În caz de eroare (dacă așa caracter nu există în șirul S) funcția întoarce valoare nulă.

Exemplu:

```
#include <string.h>
#include <conio.h>
#include <stdio.h>
void main(void)
{ clrscr();
  char S[15];
  char *ptr, c = 'r';
  strcpy(S, "TurboC++");
  ptr = strchr(S, c);
  if (ptr) {printf("Caracterul %c are pozitia %d in sirul %s\n", c, ptr-S, S);
    puts(ptr);}
  else printf("Caracterul n-a fost gasit\n");
  getch(); }
```

În rezultatul îndeplinirii acestui exemplu la monitor va fi afișat mesajul: *Caracterul r are poziția 2 în șirul TurboC++* și apoi datorită funcției *puts(ptr)*; va fi afișat șirul S trunchiat de la caracterul *r* la care indică *ptr*: „*rboC++*”.

19.Funcția strerror();

Prototip: char *strerror(int errnum);

Efect: Determină eroarea după numărul erorii și returnează un indicator la șirul de caractere ce conține descrierea erorii.

Exemplu:

```
#include <stdio.h>
#include <conio.h>
#include <errno.h>
void main(void)
{ clrscr();
  char *numerr;
```



```

numerr = strerror(11);
printf("Eroarea este: %s\n", numerr);
getch(); }

```

În rezultaul executării acestui exemplu funcția *strerror(11)*; va determina eroarea după numărul ei, iar funcția *printf* va afișa: *Eroarea este: Invalid format*.

20. Funcția *strpbrk()*;

Prototip: *char *strpbrk(const char *s1, const char *s2)*;

Efect: Funcția caută în șirul S1 primul caracter ce există și în șirul S2;

În caz de succes funcția întoarce un indicator la primul caracter din S1 apărut în S2.

Exemplu:

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void)
{ clrscr();
  char *S1="Universitate";
  char *S2="Moldova";
  char *ptr;
  ptr=strpbrk(S1,S2);
  if (ptr) printf("Primul caracter din S1 gasit in S2 este: %c\n", *ptr);
  else printf("Caracter nu a fost gasit\n");
  getch(); }

```

În rezultat va fi găsit caracterul „v”.

21. Funcția *strrchr()*;

Prototip: *char *strrchr(const char *s, int c)*;

Efect: Funcția caută ultima apariție a caracterului c în șirul S. În caz de succes funcția întoarce un indicator la ultimul caracter din S identic cu caracterul c.

Exemplu:

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void)
{ clrscr();
  char S[15];
  char *ptr, c = 'r';

```

```

strcpy(S, "Programare");
ptr = strrchr(S, c);
if (ptr) {printf("Caracterul %c este pe pozitia: %d\n", c, ptr-S);
puts(ptr);}
else printf("The character was not found\n");
getch();}

```

În rezultatul îndeplinirii acestui exemplu la monitor va fi afișat mesajul: *Caracterul r este pe poziția 8* și apoi datorită funcției *puts(ptr)*; va fi afișat șirul S trunchiat de la ultimul caracter *r* la care indică *ptr*: „*re*”.

22. Funcția *strset()*;

Prototip: *char *strset(char *s, int ch)*;

Efect: Schimbă toate caracterele din șirul S în valoarea caracterului c.

Rezultatul final se va păstra în același șir S.

Exemplu:

```

#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char s[10]="student";
printf("Șirul inițial - %s\n",s);
strset(s, 'a');
printf("Șirul final - %s\n",s);
getch();}

```

În rezultat va fi primit: *Șirul inițial – student, Șirul final – aaaaaaa*.