Sergiu G. Istrati

P R O G R A M A R E

Iniţializare în limbajele C şi C++

Ciclu de prelegeri la disciplina "Programare"

Chişinău 2003

Adnotare

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

Autor: lector superior Sergiu G.Istrati

Redactor responsabil: conf. univ. dr. Pavel Nistiriuc

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

U.T.M. 2003

Cuprins

Limbajul de programare C/C++

Introducere

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

Limbajul de programare C a fost elaborat de către Denis M.Ritchi în 1972 şi descris detaliat în cartea "Limbajul de programare C" de Ritchi şi Brian B.Kernigan. Realizarea limbajului în conformitate cu regulile descrise în carte poartă denumirea de "Standard C K&R" şi 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 aşa fel limbajul C++ posedă marea majoritate a posibilităţilor limbajului ANSI-C şi în plus la acestea alte instrumente de programare cu posibilităţi mai avansate.

Lucrarea de faţă conţine descrierea atît a instrumentelor din limbajul ANSI-C ce sînt susţinute şi de copmpilatoarele C++, cît şi descrierea instrumentelor de programare ale limbajului C++.

Deasemenea în lucrare este atrasă o deosebită atenţie 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 studenţilor 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 ştim deja o numim algoritm. Întreaga teorie informatică se ocupă de fapt cu elaborarea unor noi calculatoare, limbaje de programare şi algoritmi. Datoria oricărui limbaj de nivel înalt este să ne pună la dispoziție o sintaxă cît mai comodă prin care să putem descrie datele cu care lucrează programul nostru şi instrucțiunile care trebuiesc executate pentru a rezolva o anumită problemă.

Limbajul de programare C ca şi orice alt limbaj de programare îşi are alfabetul său şi specificul de utilizare a simbolurilor. Alfabet al unui limbaj de programare se numeşte un set de simboluri permis pentru utilizare şi recunoscut de compilator, cu ajutorul căruia pot fi formate mărimi, expresii şi operatori ai acestui limbaj de programare. Alfabetul oricărui limbaj de programare conține cele mai simple elemente cu semnificație lingvistică, iar sintaxa limbajului defineşte modul în care se combină elementele vocabularului pentru a obține fraze corecte (instrucțiuni, secvențe de instrucțiuni, declarări 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 şi 127, numit cod ASCII. Mulțimea elementelor alfabetului limbajului de programare C se poate împărți în 5 grupe:

1) Simboluri folosite la formarea identificatorilor şi cuvintelor cheie. În componența aceastei grupe intră literele minuscule şi majuscule ale alfabetului latin (englez) şi simbolul subliniere "_". Este de menționat faptul că literele minuscule şi majuscule de acelaşi fel (Exemplu: a şi 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 şi majuscule ale alfabetului rus (sau altui alfabet național) şi 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 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 le organizarea proceselor de calcul şi la transmiterea compilatorului unui set de instruncțiuni.

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

5) Pe lîngă grupele de simboluri precăutate limbajul C pe larg foloseşte consecutivități de conducere, adică combinații de simboluri speciale folosite în funcțiile de intrare-ieşire 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 şi cifre.

Consecutivitatea de conducere	Denumirea	Echivalentul hexazecimal
\a	Sunet (beep)	007
\b	Întoarcere cu o poziție	008
\t	Tabulare orizontală	009
\n	Trecere în rînd 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 şi \xddd (aici prin d este notată o cifră orecare) 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, intrrupt, 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 instrucţiunilor din C. Cuvintele cheie ce încep cu semnul _ (subliniere) se folosesc pentru acces la segmentele de date şi la registrele calculatorului.

Prin sintaxa unui limbaj de programare se înţelege, în general, un ansamblu de reguli de agregare a unităţilor lexicale pentru a forma structuri mai complexe (instrucţiuni, declaraţii, programe etc.) Structura unui program în limbajul de programare C deasemenea îşi are regulile ei de sintaxă după cum urmează: un antet, urmat de funcţiile secundare după care urmează corpul funcţiei principale (care conţine în componenţa sa şi o parte declarativă). Şi pentru descrierea în detaliu a acestor componente sunt necesare, desigur şi 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 şi e necesar să alcătuim un program, care calculează şi afişază suma elementelor întregului masiv. Aceasta este o problemă tipică pentru prelucrarea masivelor unidimensionale.

În concepţiile limbajului C fiecare algoritm evidenţiat se realizează de o unitate de program numită funcţie. Stilul de programare în C se caracterizează prin tendinţa de a evidenţia un număr mare de funcţii nu prea voluminoase, astfel, ca prelucrarea datelor în aceste funcţii să nu depindă de celelalte părţi a programului. Acest lucru face programul destul de înţeles şi dă posibilitatea de a introduce uşor corecţii în unele funcţii fără a tangenta celelalte. Prin funcţiile evidenţiate unica cu care începe îndeplinirea programului se numeşte principală şi are denumirea fixată: main. Toate celelalte funcţii au denumire arbitrare, pe care programatorul singur le numeşte. Ele pot fi înscrise în fişierul iniţial pînă la funcţia main (în ordine arbitrară) sau se pot afla în fişiere diferite pe un suport magnetic. În programul de mai jos, ce realizează rezolvarea problemei intră funcţia main() şi funcţia 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>* şi *#include <conio.h>* nu sînt instrucţiuni 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 înţeles că în fişierul ce se compilează trebuie introdusă informaţia din fişierele sistemului Turbo C *stdio.h*

(Standard Input/Output Header – titlu de introducere-extragere) şi *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()* şi *clrscr()*, informația despre care se conține în fişierele indicate. Urmatorul rînd „*int suma (int y[10], int m)*“ conține declararea functiei *suma()* cu 2 parametri: un masiv *y[10]* şi o variabila simpla *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)* defineşte 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 urmeaza după *main()* indică compilatorului că aceasta este o funcție, iar cuvîntul *void* din paranteze – faptul că funcția nu foloseşte parametri. Perechea acoladelor: prima ce se deschide după *main()* şi corespunzător acolada ce se închide după funcția *getch();* – mărgineşte instrucțiunile care formează corpul funcției principale *main()*. În limbajul C perechea de accolade *{}* mărgineşte 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* şi *i* de tip întreg şi 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 afişază pe ecran un comentariu. În acest caz funcția *printf("Culege marimea masivului n<10\n");* afişază pe ecran propunerea de a culege mărimea masivului şi simbolul *\n* de trecere în alt rind. Funcția următoare *scanf("%d", &n);* este o funcție de intrare şi face posibilă introducerea de la tastatură a valorii mărimii 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 afişază un comentariu pe ecran şi propune culegerea valorilor elementelor masivului *x* cu marimea *n* deja cunoscută. Următoarea instrucțiune este instrucțiunea ciclică *for*. Această instrucțiune este compusă şi 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-şi schimbă valoarea de la *0* la *n* cu pasul 1 datorită instrucţiunii de incrementare *i++*. Corpul ciclului, care va fi repetat de *n* ori este mărginit de perechea de accolade deschisă şi închisă, şi este compus din 2 funcţii: prima *printf(),* care afişază un comentariu pentru culegerea valorii elementului current al masivului şi a doua *scanf(),* care face posibilă înscrierea valorii elementului current al masivului de la tastatură în memorie. În aşa fel, la sfîrşitul îndeplinirii ciclului toate elementele masivului *x[n]* vor avea valori, fapt ce face posibilă calcularea sumei totale a elementelor din masiv. După executarea instrucţiunii ciclice *for* urmează instrucţiunea de atribuire *w=suma(x,n);*. În partea dreaptă a acestei atribuiri este amplasată funcţia *suma()*. Anume valoarea acestei funcţii, după executarea cu folosirea parametrilor locali *x* şi *n*, va fi atribuită variabilei *w*. Să analizăm funcţia *suma():*

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

Ca şi la declararea oricărei funcţii în limbajul C întîi de toate urmează antetul funcţiei: *int suma (int y[10], int m)* , unde cuvîntul int din faţa numelui funcţiei suma este tipul valorii întors de funcţie în programul principal *main()*. Parantezele rotunde după numele funcţiei mărginesc lista de parametri formali folosiţi de funcţie. Aceşti parametri sînt un masiv de tip întreg *y* cu lungimea 10 elemente şi o variabilă de tip întreg *m*. Este de menţionat faptul că la momentul chemării funcţiei *suma()* din cadrul programului principal *main(),* valorile parametrilor actuali (în cazul nostru *x*-masivul prelucrat şi *n*-mărimea lui) sînt atribuite parametrilor formali din funcţie (în cazul nostru masivul *y* şi marimea lui *m*) şi este necesară îndeplinirea următoarei condiţii: parametrii actuali trebuie să corespundă ca cantitate, poziţie şi tip cu parametrii formali. Acoladele deschisă după antetul funcţiei şi corespunzător închisă după operatorul *return();* delimitează corpul funcţiei *suma(),* care îşi are şi ea secţiunea de declarare a datelor. Aici *int i, suma=0;* declară variabila locală *i* de tip întreg şi iniţializează funcţia *suma()* cu valoarea 0. Următoarea este instrucţiunea ciclică *for* care conţine în corpul său delimitat de accolade o singură instrucţiune: instrucţiunea compusă de atribuire *suma+=y[i];* echivalentă cu instrucţiunea *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 conţine valoarea finală a sumei tuturor elementelor din masiv. Transmiterea acestei valori funcţiei principale *main()* este efectuată de către operatorul *retun(suma);* . După executarea acestui operator valoarea sumei va fi

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

Deci, valoarea sumei elementelor masivului va fi atribuită variabile *w*. După aceasta urmează afişarea pe ecran a rezultatului final: *printf("suma=%d\n",w);* . Ultima instrucţiune din program este apelul la funcţia *getch(),* care opreşte executarea programului cu scopul vizualizării rezultatului pînă cînd nu va fi culeasă tasta *Enter.*

În aşa fel poate fi descrisă structura generală a unui program în C după cum urmează: orice program începe cu includerea bibliotecilor de funcţii care vor fi folosite în program, după aceasta urmează declararea tuturor funcţiilor auxiliare folosite în program, care au următoarea componenţă: antetul funcţiei, secţia de declarare a variabilelor, constantelor locale, după care urmează corpul funcţei; după declararea tuturor funcţiilor auxiliare urmează corpul funcţiei principale *main()* delimitat de o pereche de accolade, care conţine descrierea variabilelor, constantelor şi însăşi instrucţiunile programului principal.

3. Tipuri de date.

Un program în limbajul C conţine o descriere a acţiunilor ce trebuie să fie executate de calculator şi o descriere a datelor ce sunt prelucrate de aceste acţiuni. Acţiunile sînt descrise prin instrucţiuni, iar datele prin declaraţii (sau definiţii).Prin tip de date se înţelege o mulţime de valori care pot fi atribuite unei variabile sau constante. Tipurile de date în C pot fi împărţite în două categorii: simple (elementare) şi compuse (structurate). În general, tipurile de date sunt definite explicit de către programator şi sunt specifice programului în care apar. Există însă tipuri de date elementare de interes mai general, numite tipuri predefinite a căror definiţie se consideră cunoscută şi nu cade în sarcina programatorului.

3.1. Tipuri de date simple predefinite. Constante.

Un program în C conţine în mod explicit diferite valori textuale şi numerice. Aşa valori, ce apar în program, sînt numite constante. Constanta – este o valoare numerică sau textuală, care întotdeauna este definită şi în mersul îndeplinirii programului rămîne neschimbată. Tipul constantei se defineşte de forma ei de înscriere, iar valoarea ei este încheiată în ea înseşi.

3.1.1. Constante întregi.

Constanta întreagă este un număr înscris în program fără punct zecimal şi 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 şi prima cifră nu e zero, atunci constanta se consideră zecimală. De exemplu: 123, 45, 37. Dacă constanta este înscrisă folosind cifrele 0..7 şi prima cifră este zero, atunci constanta se consideră octală. De exemplu: 045, 037. Dacă constanta este înscrisă cu ajutorul cifrelor 0..9 şi literelor a..f sau A..F şi se începe cu 0x sau 0X, atunci constanta este hexazecimală. De exemplu: 0x45, 0x37. În aceste exemple constantele înscrise cu unele şi aceleaşi 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 şi 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 imt)	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 rămaşi definesc valoarea ei. În aşa 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. Aşa 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 rămaşi – ca valoare a numărului. Aşa 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. Aşa 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 şi 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 şi de a indica interpretarea lor ca fără semn (*unsigned*). Pentru aceasta se folosesc modificatori speciali, înscrişi 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 şi 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 şi valoarea constantei este necesară prezența la minim un spațiu. În exemplele de mai sus compilatorul implicit va atribui variabilelor *K* şi s*alariu* valori de tipul întreg, întrucît numerele 35 şi 700 au sintaxa şi 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 foloseşte cuvîntul rezervat *const* pentru inițializarea constantei, descrierea tipului ei şi atribuirea valorii şi 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 şi *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 şi formă cu ordin. Constanta reală în formă cu punct se scrie ca fracție zecimală cu semn sau fără semn, la care partea întreagă şi partea fracționară se despart prin punct. Dacă semnul constantei este omis ea se socoate pozitivă. Constanta reală în formă cu ordin este comodă pentru scrierea numerelor foarte mari sau foarte mici. În C, ca şi în majoritatea limbajelor de programare, pentru aşa înscrieri se folosesc constante reale în formă cu ordin, ce au

aspectul: *mantisa*_e_*ordinul* sau *mantisa*_E_*ordinul*. În aceste notaţii 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 lipseşte se subînțelege semnul +. De exemplu *7.32E+14 = 7.32*10*^{14}*; 55.000000E-3 = 0.055;*

Pentru determinarea constantelor de tip real sunt folosite diferite cuvinte rezervate, care determină diapazonul valorilor şi 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 orecare de aproximaţie, care depinde de compilator. În aşa fel cifra 6.12345678912345 în diapazonul de aproximaţie pentru tipul *float* va fi interpretată de compilator ca 6.123456, acest tip se mai numeşte tip cu aproximaţie unitară şi are aproximaţia –6 poziţii după punct. Tipul double se mai numeşte tip cu aproximaţie dublă şi are aproximaţia de pînă la 15 –16 poziţii după punct. Sintaxa de iniţializare 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 cîte un octet. Mulţimea valorilor unei constante de tip carecter este următoarea: literele minuscule şi majuscule ale alfabetului latin, zece cifre arabe 0..9 şi 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 faţa 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 întîlnesc, 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. Şiruri de caractere.

Un şir 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 şirurile de caractere. Limbajul C operează cu şirurile cum ar lucra cu o succesiune de date de tip caracter amplasate într-o structură numită masiv. Aici fiecare simbol din şir este o componentă aparte a masivului de tip char. Tipul masiv este un tip structurat în C şi va fi studiat în capitolul "Tipuri structurate"

Notă: Unele compilatoare C şi C++ susțin tipuri de date speciale pentru operare cu şiruri de caractere şi conțin biblioteci de funcții pentru prelucrarea şirurilor.

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 şi simplu nume.

4.1. Nume de variabile (identificatori).

Numele variabilei (identificatorul variabilei) este o succesiune de caractere şi 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 şi 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, a*1, 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 înscriere: *tip v1,v2,...,vn;*

unde : *tip* - determină tipul valorilor pe care le pot primi variabilele v1,v2,..,vn. În calitate de tip se foloseşte unul din cuvintele cheie deja cunoscute int, float, double sau char. Punctul şi virgula este semnul sfîrşitului instrucțiunii. Între cuvîntul cheie ce determină tipul variabilelor şi lista numelor variabilelor, trebuie să fie cel puțin un blanc. 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 numeşte 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 sun egale cu *1,7e-12*. Din aceatsă cauză e mai bine de evitat amestecarea variabilelor inițializate şi neinițializate în una şi aceiaşi instrucțiune de descriere.

Există cazuri, cînd 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. Cînd variabila este declarată şi ei i se rezervează loc în memorie, conținutul acestor celule de memorie nu se schimbă pînă cînd variabila nu este inițializată sau pînă cînd ei nu i se atribuie vre-o valoare. Din această cauză, dacă variabila ce n-a fost inițializată şi 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 şi ' ' (spațiu) pentru cele caracteriale sau de tip şir de caractere.

5. Operaţii şi expresii.

5.1. Operaţii aritmetice.

În calitate de operaţii aritmetice limbajul C foloseşte următoarele:

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

5.2. Operaţia de atribuire.

Operaţia de atribuire poate fi simplă sau compusă, în dependenţă de semnul operaţiei de atribuire. Operaţia de atribuire simplă se notează cu semnul = şi 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 operaţii de atribuire. Operaţiile de atribuire au o prioritate mai joasă decît cele descrise anterior şi se îndeplinesc de la dreapta spre stînga. Operaţia de atribuire leagă variabila cu expresia, atribuind variabilei valoarea expresiei, ce stă în dreptul semnului de atribuire.

Operaţiilor binare*: + - * / % >> << & | ^* le corespunde operaţia de atribuire compusă, care se notează ca *op=,* unde op este una din operaţiile enumerate mai sus, de exemplu, += şi *= etc.

Operaţia de atribuire compusă sub forma: _variabila_op=_expresie_ se defineşte prin operaţia de atribuire simplă astfel: _variabila_=_variabila_op_expresie_

De exemplu, *d+=2* este echivalent cu *d=d+2*. De regulă pentru operaţiile de atribuire compuse calculatorul construieşte un program mai efectiv.

5.3. Operaţii de incrementare(decrementare).

Operaţia de incrementare notată prin două semne plus consecutive, se aplică asupra unei singure variabile şi măreşte valoarea cu o unitate. Operaţia de decrementare se notează prin două semne minus micşorînd valoarea ei cu o unitate. Operaţia de incrementare poate fi folosită atît ca prefix cît şi ca sufix. Diferenţa dintre acestea constă în momentul schimbării valorii variabilei. Forma prefixă asigură schimbarea variabilei pînă la folosirea ei. Iar forma sufixă - după folosirea ei. Operaţiile de incremenatare decrementare se aplică asupra unor variabile de tip ordonat(aşa ca *int*). Exemple:

Operaţia *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; întîi se va incrementa x şi numai după aceia va fi calculată valoarea pentru s.

Şi 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, şi doar apoi variabila m se va mări cu unu. Expresia *(i+j)++* este greşită căci operația de incrementare se aplică doar către numele variabilei. Operația de incrementare are o prioritate mai înaltă decît cea aritmetică.

5.4. Relații şi 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 decît cele aritmetice şi de incrementare. Între operațiile de relație primele patru au aceiaşi prioritate şi se îndeplinesc de la stînga la dreapta. Ultimele două operații au o prioritate încă mai mică şi expresia a va primi valoarea unu doar atunci cînd şi prima expresia şi a doua expresie este adevărată. Operațiile de relație se pot folosi pentru toate tipurile de date de bază, cu excepția şirurilor de caractere (Pentru compararea şirurilor 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 şi 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* şi *expr2* sunt careva expresii , atunci

expr1 && expr2 e adevărată în cazul cînd ambele sunt adevărate

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

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

Noi am stabilit pentru *expr1* şi 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 trateaază ca adevăr iar cea nulă ca fals. Prioritățile operațiilor logice se aranjeaază în ordine descrescătoare astfel: *Negare (!). Conjuncție (&&). Disjuncție (||).* Operațiile logice au o prioritate mai mică decît operațiile de relație.

Calcularea expresiilor logice, în care sunt numai operații &&, se termină dacă se descoperă falsitatea aplicării următoarei operații &&. Reeşind din definiția semanticii operația &&, cum numai apare valoarea fals prelungirea calculelor 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* şi *double*.

Operațiile logice poziționale:

Numele operației logice poziționale	**Semnul operației în Turbo C**
ŞI pozițional	&
Sau pozițional	|

Sau exclusiv poziţional	^
Deplasare la stînga	<<
Deplasare la dreapta	>>
Inversare	~

Operaţia poziţională ŞI se realizează asupra fiecării perechi de biţi a datelor, de exemplu dacă în program avem descrierea variabilelor : *int n,d;* Şi 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 operaţiei poziţionale ŞI va fi :

n 0000000000000101
d 0000000000000110
4 0000000000000100

Operaţia poziţională ŞI deseori se foloseşte pentru evidenţierea vreunui grup de biţi.

Operaţia poziţională SAU se aplică asupra fiecărei perechi de biţi a datelor şi deseori se foloseşte pentru instalarea unor biţi, de exemplu, expresia: *X=X | mask* instalează în unu acei biţi ai lui x, cărora le corespunde unu în mask. Nu trebuie să confundăm operaţile logice poziţionale cu operaţiile logice *&&* şi || . De exemplu, expresia *1&2* are valoarea 0, iar expresia *1&&2* are valoarea 1.

SAU exclusiv poziţional realizează asupra fiecărei perechi de biţi operaţia de adunare după modulul 2. Rezultatele îndeplinirii operaţiei SAU exclusiv poziţional 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

Operaţiile de deplasare la sănga(>>) şi la dreapta (<<) îndeplinesc deplasarea primului operand la stînga sau la dreapta cu un număr de poziţii binare determinat de operandul al doilea.

În timpul deplasării la stînga biţii, ce se eliberează, se completează cu zerouri. De exemplu, x<<2 deplasează x la sînga cu două poziţii, completînd poziţiile eliberate cu zerouri,ceea ce-i echivalent cu înmulţirea la 4. În caz general, a deplasa x cu n poziţii la stînga e echivalent înmulţirii valorii lui x cu 2 la puterea n.

Deplasarea la dreapta a unei mărimi fără semn (unsigned) aduce la completarea cu zerouri a biţilor eliberaţi. Deplasarea la dreapta a unei mărimi cu semn la unele mşini aduce la multiplicarea bitului semnului, iar la altele – biţii se completează cu zerouri. Turbo C multiplică bitul semnului şi de aceea pentru nemerele pozitive, deplasarea la dreapta cu n poziţii e echivalentă împărţirii la 2 la puterea n.

Operaţia unară inversare (~) [tilda] transformă fiecare poziţie unară în nulă şi invers fiecare poziţie nulă în unară.

5.6. Operaţia dimensiune.

Operaţia unară dimensiune, notată prin cuvîntul-cheie *sizeof*, dă mărimea operandului său în octeţi. Operaţia dimensiune se foloseşte sub forma: *sizeof_expresie* sau *sizeof_tip*

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

Operaţia *sizeof* poate fi folosită peste tot, unde se admite folosirea unei constante întregi. Construcţiile s*izeof_expresie* şi *sizeof_tip* se cercetează ca ceva unitar şi, 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*, şi nu *a+b+c+d.*

5.7. Operaţia virgulă.

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

De exemplu: *k=a+b,d=m+n,5.2+7* este o expresie cu virgulă şi se calculează de la stînga spre dreapta. Valoarea ei este 12.2 de tip *float*. În procesul calculării acestei expresii variabilelor *k* şi *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 foloseşte în două contexte: ca separator a datelor şi ca operație, ce determină calculul consecutiv al expresiilor. De aceea e admisă, de exemplu, aşa o expresie: *int a,b,c=(1,2,5),d;*
unde variabila *c* se inițializează cu o expresie constantă cu virgulă 1,2,5 şi primeşte 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:

expr1_?:expr2_:expr3_;

Valoarea expresiei condiționale se calculează astfel: prima se calculează *expr1_*. Dacă ea este diferită de zero (e adevărată), atunci se calculează *expr2_* şi valoarea ei va fi valoarea întregii expresii, în caz contrar se calculează *expr3_*. Astfel, semnele de operație *?* şi *:* determină operația ternară, adică operație cu trei operanzi. De exemplu, pentru a calcula z maximum din a şi b e suficient de a scri 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. Totuşi, se recomandă de a pune parantezele, căci în aşa 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 şi constantelor de acelaşi 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 îndeplineşte asupra datelor de diferite tipuri, atunci ambele date se reduc la cel mai superior din aceste două tipuri. O astfel de conversie se numeşte ridicarea tipului. O consecutivitate de tipuri ordonate de la superior la inferior e determinată în corespundere cu prezența interioară a datelor şi arată astfel : double, float, long, int, short, char. Modificatorul unsigned măreşte 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 micşorează greşeala de rotungire. 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; şi avem expresia: *a*b+c*

La calcularea valorii expresiei variabilele *a, b* şi *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 paranezele din fața expresiei. În această operație în calitate de _tip_ pot fi folosite aceleaşi cuvinte cheie, ca şi în instrucțiunile de descriere a tipului, împreună cu modificatorii admisibili.

Să cercetăm două expresii în Turbo C.

d=1.6+1.7 şi *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 şi 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 truchierii 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 foloseşte 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 patrate 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 rămîne de tip *int*.

5.10. Prioritățile operațiilor.

Priorităţiile operaţiilor se cercetau în procesul expunerii lor. În paragraful de faţă sunt reprezentate operaţiile descrise mai sus şi e indicată ordinea îndeplinirii lor. Pentru compararea priorităţilor operaţiilor le prezentăm în tabel în ordinea descreşterii priorităţii. Coloana tabelului "Ordinea execuţiei" determină consecutivitatea îndeplinirii pentru operaţiile cu aceiaşi prioritate. De exemplu în expresia *k=d+=b-=4;* consecutivitatea îndeplinirii operaţiilor se va determina în ordinea de la dreapta la sînga şi în rezultat b se va micşora cu 4, d se va mări cu *b-4*, *k=d+b-4*. Ordinea de evaluare, descrisă de o simplă operaţie de atribuire, va fi următoarea: b=b-4; d=d+(b-4*); k=d;*

Prioritatea	**Semnul operaţiei**	**Tipul operaţiei**	**Ordinea execuţiei**
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	< > <= >=	Relaţii	
7	== !=	Relaţii (egalitate)	
8	&	ŞI poziţional	
9	^	SAU exclusiv poziţional	
10	|	SAU poziţional	
11	&&	ŞI logic	
12	||	SAU logic	
13	?:	Condiţie	
14	= *= /= %= += -= &= |= >>= <<= ^=	Atribuire simplă şî compusă	De la dreapta la stînga
15	,	Virgulă	De la stînga la dreapta

6. Instrucțiuni.

Prin instrucțiune în C se subînțelege o oarecare înscriere, ce se termină cu caracterul; (punct şi virgulă), sensul căreiea determină acțiunile compilatorului în timpul prelucrarii textului inițial al programului sau actiunile 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 algorilmul dat. Instrucțiunile de compilare au fost precautate în compartimentele de descriere şi declarare a datelor. În continuare vor fi studiate instrucțiunile de îndeplinire.

Fiecare instrucțiune are sintaxa şi 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 şi blocuri.

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

Bloc – se numeşte un şir 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 aşa caz se spune, că aceste variabile sunt localizate în interiorul blocului, există şi lucrează numai în interiorul blocului şi se pierd în afara lui.

În Turbo C instrucțiunea compusă şi 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 şi alte blocuri.

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

Eticheta se foloseşte doar în cazul cînd se foloseşte saltul necondiționat la aceaastă 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 aşa o instrucțiune ce nu îndeplineşte nici o acțiune. În cazul cînd avem lipsa instrucțiunii şi 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 sfîrşitului 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 aşa fel, instrucțiunea-expresie va avea următoara formă*: expresie_;*

După cum sa menționat, orice instrucțiune poate fi etichetată. Instrucțiunea-expresie aderă la clasa instrucțiunilor simple ai limbjului 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 şi atunci instrucțiunea-expresie în limbajul Turbo C are acelaşi sens şi 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).

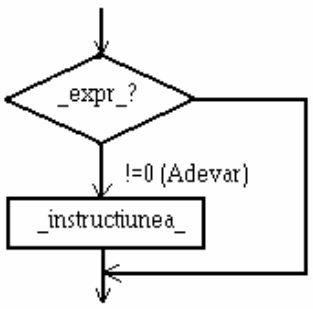
6.3.1. Instrucțiunea de ramificare IF şi IF-ELSE.

În structurile ramificate de calcul, unele etape nu întotdeauna se îndeplinesc în una şi aceiaşi 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* şi *if – else*. Instrucțiunea *if* e compusă şi sintaxa ei admite unul din următoarele formate:

if_(expresie)_instrucțiune sau

if_(expresie)_instrucțiune1_else_instrucțiune2;

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



1. Formatul *if(expresie)_instrucțiune*. Mai întîi se calculează valoarea expresiei. Dacă rezultatul ei este ADEVĂR (adică expresia != 0), atunci se îndeplineşte _instrucțiunea, iar în caz contrar se sare peste instrucțiune şi 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 şi c rămîn neschimbate. Instrucțiunea de ramificare a acestui caz:

if(d==c) ++d,c+=3;

În calitate de *instrucțiune_* aici se foloseşte 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* şi 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, şi numai *c* se măreşte cu 3.

Vom considera acest exemplu din nou:

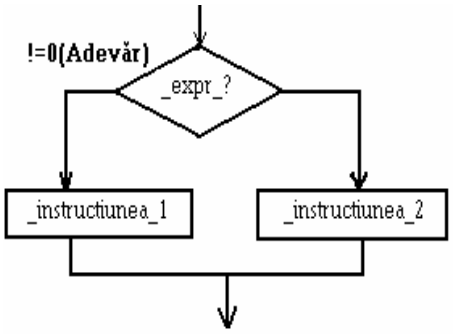
if(d= =c){++d;c+=3;}

În cazul că *(d == c)* va avea valoarea FALS, *d* şi *c* vor rămîne neschimbate, deoarece aceste instrucţiuni sînt incluse în acolade, adică formează un bloc, şi din punct de vedere logic sunt privite ca o singură instrucţiune bloc.

2. Formatul *if(expresie)_instucţiune1_else_instrucţiune2.*

Ca şi în cazul precedent, în primul rînd se calculează valoarea expresiei. Dacă ea diferă de zero, adică este ADEVĂR, atunci se îndeplineşte instrucţiunea1, în caz contrar se îndeplineşte instrucţiunea2.

Schematic această instrucţiune e preszentată pe desen.



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

Atunci putem scrie :

if(a>b)z=a;else z=b; Prezenţa (;) după *z=a* este necesară, deoarece aceasta este o instrucţiune ce intră în coponenţa instrucţiunii *if*.

Prima formă prescurtată a instrucţiunii *if* ne dă posibilitatea îndeplinirii sau neîndeplinirii oricărei operaţii, pe cînd a doua formă oferă posibilitatea alegerii şi îndeplinirii unei operaţii din două posibile. Totuşi, cîte odată apare necesitatea alegerii unei operaţii din mai multe.Vom examina un lanţ de instrucţiuni:

if(expresie1)_instrucţiune1_else_if(expresie2)_instrucţiune2_else_if(expresie3) _instrucţiune3_else _instrucţiunea4…

Aşa un lanţ de instrucţiuni e admis, deoarece în locul instrucţiunii în instrucţiunea if poate fi orice instrucţiune, inclusiv şi if. Însă o aşa înscriere e complicată. Pentru aşa procese în C există o instrucţiune 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 instrucţiunea *if* de formă scurtă, care are ca instrucţiune forma *if_else;* . În cazul b) avem instrucţiunea *if_else* în formă plină, avînd în calitate de *instrucţiune1* forma *if* prescurtată. Secvenţa b diferă de a numai prin prezenţa acoladelor, ce definesc un bloc, însă, evident, joacă un rol important la interpretarea acestor intrucţiuni.

6.3.2. Instrucţiunea de salt necondiţionat GOTO.

Instrucţiunea *goto* ne dă posibilitatea transmiterii controlului execuţiei programului la o instrucţiune marcată cu o etichetă. Instrucţiunea *goto* are formatul:

goto_etichetă;

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

Exemplu :Folosirea instrucțiunilor if şi goto la organizarea ciclurilor. Acest program calculează valoarea lui y care este egală cu $\sum n/(n+5)$,unde n=1,..,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* şi *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* şi *goto*, dar în nici un caz nu serveşte 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.

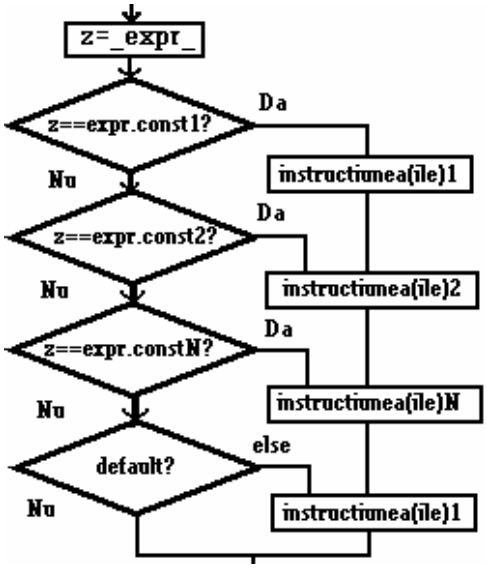
Instrucțiunea de selectare e destinată pentru selectareaa unei singure variante dintr-o mulțime posibilă de ramificări în program. Orice ramificare poate fi organizată cu ajutorul lanțului de instrucțiuni *if_else_if_..._else_,* însă în cazurile cînd sunt prezente mai mult de 3 condiții e mai rațională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

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

case expr_const_n:instrucţiune_n;
default:instrucţiune; }

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 instrucţiunile – varantele 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, prezenţa careia nu este obligatorie, dar în majoritatea cazurilor acest prefix este prezent în construcţia *switch*.

Instrucţiunea 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 şi se îndeplineşte variantaa i, pentru care aceste valori coincid. După ce sau îndeplinit instrucţiunile ce se conţin în case-ul respectiv, se îndeplinesc instrucţiunile variantei următoare şi aşa pînă ce se termină toate instrucţiunile variantei. Dacă valoarea expresiei în switch nu coincide cu nici o valoare a case prefixelor, atunci ori se îndeplineşte instrucţiunea variantei cu prefixul default (dacă default este prezent), ori nu se îndeplineşte nici o variantă (dacă default este absent). Schema algoritmică de îndeplinire a instrucţiunii switch este prezentă pe desen.



Să examinăm exemplul de întrebuinţare a instrucţiunei switch : se scrie un fragment de program, ce tipăreşte patru rînduri din poezie, începînd de la rîndul k:

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

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

"Din rude mari împărăteşti
O preafrumoasă fată.
Poezia nu conţine rînd cu aşa număr "

6.3.4. Instrucțiunea de întrerupere BREAK.

În practica programării cîteodată apare necesitatea îndeplinirii numai a unei variante case fără indeplinirea celor ce urmează după ea, adică trbuie întreruptă logica stabilită de lucrul a instrucțiunii switch. Evident că pentru aşa ceva e necesar inroducerea unei instrucțiuni între instrucțiunile variantei, care ar servi ca sfîrşit al îndeplinirii lui switch. Aşa 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 cînd 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 scri instrucțiunea *BREAK* după varianta default. Default poate fi situat în orice loc al instrucțiunii switch, chiar şi pe primul loc între variante, şi 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 poveşti,\n");break;
case 2 : printf("A fost ca niciodată,\n"); break;
case 3 : printf("Din rude mari împărăteşti,\n"); break;
case 4 : printf("O preafrumoasă fată.\n"); break;
default:printf("Poezia nu conține rînd cu aşa număr\n"); }

Trebuie de menționat, că instrucțiunile din variantele *case* sau *default* pot lipsi. Aceasta este necesar cînd avem nevoie de a obține acelaşi 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 caszul cînd L=`C`, atunci va fi îndeplinită instrucțiunea ce urmează după primul case, dar deoarece aici lipseşte instrucțiunea, se vor îndeplini instrucțiunile situate mai jos, pînă ce nu se întîlneşte instrucțiunea break.

6.4. Instrucțiuni iterative(ciclice).

Instrucțiunile precăutate mai sus redau operații care trebuie efectuate conform algoritmului şi fiecare din ele se îndeplinesc numai odată. În cazul, când una şi aceiaşi 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)

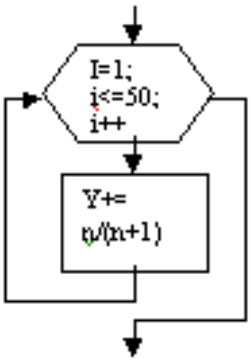
6.4.1. Instrucțiunea ciclică FOR.

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 primeşte 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 îndeplineşte instrucțiunea. După îndeplinirea instrucțiunei se execută expresia de corecție şi din nou se controlează expresia de control, justețea căreia duce la îndepliniea 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=Σi/(i+1); unde i=1..50;

y=0; for(i=1;i<=50;i++) {y+=i/(i+1) };

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



Schema algoritmică îndeplinrii instrucțiuni 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 şi evident, se poate întîmpla 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 restrucţii 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.

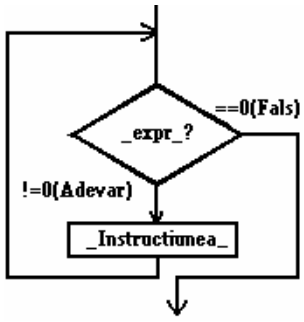
6.4.2. Instrucţiunea ciclică WHILE.

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 aceiaşi), atunci instrucţiunea se îndeplineşte o dată şi apoi expresia din nou se testează.

Această succesiune de acţiuni, ce constă în testarea expresiei şi îndeplinirea instrucţiunei, se repetă periodic pănă 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 instrcţiuni. Schema algoritmică a executării instrucţiuniii while e prezentată pe desen.



Se observă, că instrucţiunea while este un ciclu cu precondiţie. Testul de control este executat înaintea 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 şi de tip float, adică pasul ciclului poate fi diferit de 1 şi chiar un număr fracțoinar. Exemplu*:*

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

Ca şi în cazul ciclului for, corpul unui ciclu while poate fi deasemenea un ciclu. Exemplu:

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

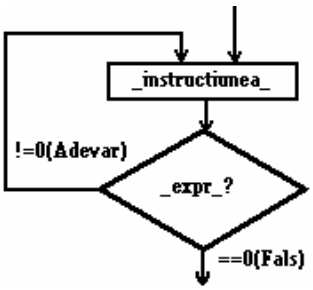
6.4.3. Instrucțiunea ciclică DO_WHILE.

Instrucțiunea ciclică *DO_WHILE* se foloseşte în cazul cînd numărul de repetări ale ciclului nu-i cunoscut, dar în acelaşi 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 îndeplineşte instrucțiunea, apoi se calculează valoarea expresiei. Dacă valoarea expresiei este adevărată, atunci instrucțiunea se îndeplineşte din nou, dacă expresia este falsță, atunci îndeplinirea ciclului se termină.

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



Exemplu: *i=1;do {y+=i(i+1); i++;} while(i<=50);*

Ciclul do_while deasemenea 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 şi DO_WHILE le putem folosi în diferite scopuri aparte sau împreună pentru cazuri concrete. În cazul cînd cunoaştem 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*. Şi 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 atît, 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, cînd la îndeplinirea a careva condiții trebuie de întrerupt executarea iterații curente şi de trecut la îndeplinirea iterației următoare a ciclului. În aşa 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 şi în instrucțiunea switch. Ea serveşte la depăşirea 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 precăutăm următorul exemplu:

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

În cazul, cînd 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 şir ordonat de elemente de acelaşi tip. Faptul, ca masivul este un tot intreg, compus din cîteva 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, adica tipul masivului. Tip al unui masiv poate fi orice tip de date deja definit: întreg, real, caracterial ş.a.

nume este numele masivului. In calitate de nume al masivului este folosit orice identificator. Mai mult ca atît, deoarece numele masivului este identificator, asupra lui se răspîndeşte 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 şir de elemente aranjate uniform într-un rînd. Fiecare element al unui masiv unidimensional are 1 coordonată: numărul de ordine a elementului în şir.

2. masive bidimensionale (cu 2 dimensiuni);

masivul bidimensional reprezintă o structură formată din rînduri şi coloane. Fiecare element al unui masiv bidimensional are 2 coordonate: numarul rîndului şi numărul coloanei.

3. masive tridimensionale (cu 3 dimensiuni);

masivul tridimensional reprezintă o structură echivalentă cu un cub în volum cu 3 dimensiuni: lungimea, lăţimea, înălţimea. Fiecare element al unui masiv tridimensional are 3 coordonate: numarul rîndului(în lungime), numărul coloanei(în lăţime) şi numărul înălţimei(în adîncime).

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(propoziţie) 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 conţin elemente cu valorile cărora se operează în program. Fiecare element în masiv îşi are indicele şi valoarea sa. În calitate de indice a unui element se foloseşte 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 pîna 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 şi bidimensionale. Accesul la un element al unui masiv unidimensional se face în felul următor: *nume[i];* unde *nume* - numele masivului, *i*- numarul 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 numarul 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 şi 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 şi 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 şi accesul la un element al masivului se face conform regulilor de atribuire şi 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 numeşte 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 începînd 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 numaidecît de indicat dimensiunile masivului.Compilatorul va determina numărul elementelor după descrierea masivului şi 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 şi în cazul precedent.

Vom examena 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 şi nu pot fi schimbate în decursul îndeplinirii programului.

Să examenăm 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 acelaşi. 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 greşită. 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 şi bidimensional:

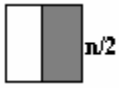
Exemplu 1. Prelucrarea unui masiv unidimensional:

Este dat un masiv unidimensional *x* cu *n* elemente. Comparați suma primei jumătăți a masivului cu media aritmetică jumatății 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 haşurată.

```
#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 haşurata este=%d k=%d\n",p,k);
getch(); }
```

8. Şiruri de caractere.

Numim şir o succesiune de caractere ale alfabetului, adică o propoziție. Spre deosebire de alte limbaje de programare, limbajul C nu contine un tip de date special ce desemnează şiruri de caractere. Limbajul C opereaza cu şirurile cum ar lucra cu o succesiune de date de tip caracter amplasate într-un masiv. Aici fiecare simbol din şir este o componentă aparte a masivului. Deci, pentru a defini o variabilă, valoarea careia va fi un şir de caractere, în limbajul C trebuie de declarat un masiv de tip char cu o lungime egală cu cantitatea maximal posibilă de caractere în şir. Următorul exemplu arată cum s-ar putea declara un asemenea masiv cu scopul de a culege un nume de la tastatură şi apoi de-l afişat 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 şi incomodități. Pentru a culege numele aici este folosit un ciclu cu 20 repetări care citeşte cîte un caracter de la tastatură şi-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* spaţii după ele. Iar afişarea numelui este făcută cu ajutorul funcţiei *printf( )* inclusă în ciclu, care afişază la monitor cîte un caracter din numele cules.

Notă: *La declararea masivului de tip char pentru descrierea unui şir se indică mărimea masivului cu o celulă mai mult de cît lungimea maximă presupusă a şirului 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 informaţiei textuale au o popularitate foarte mare, limbajul C conţine mecanisme care uşurează lucru cu şirurile de caractere. Aici şirurile sînt precăutate ca un tip special de masive, fapt ce permite întroducerea si afişarea şirurilor ca un tot întreg. Pentru întroducerea şirului în memoria calculatorului este folosită funcţia *gets( )*. Această funcţie are următoarea sintaxă:

gets (nume); unde *nume* este parametrul funcţiei şi reprezintă numele variabilei tip şir, 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 funcţia *gets( )* va precăuta primele 14 caractere culese de la tastatură ca valoare a şirului cu nume *name,* iar ultima celulă din masiv va conţine simbolul nul *"\0"*. În timpul lucrului functiei *gets( )* executarea programului se stopează. Funcţia *gets( )* aşteaptă pînă utilizatorul va culege un şir de la tastatură. Pentru ca textul cules să fie atribuit ca valoare unei variabile de tip şir, utilizatorul trebuie să culeagă tasta ENTER. După aceasta propoziţia culeasă va deveni valoare a variabilei de tip şir, iar cursorul va trece în rînd nou pe monitor. Anume in timpul culegerii tastei ENTER compilatorul C adaogă la sfîrşitul şirului simbolul nul. În exemplu de mai sus culegerea şirului se precaută ca culegerea unei variabile de tip şir aparte şi nu ca culegerea a mai multor variabile de tip caracter. Însă afişarea numelui rămîne incomodă. Aici, dacă vor fi culese mai puţin de 15 simboluri de la tastatură, atunci elementele masivului *name[15]* ce urmează după simbolul nul vor conţine mărimi aleatoare.

În afară de funcţia *gets()* limbajul C mai conţine o modalitate de a întroduce o propoziţie în memorie ca valoare a unei variabile de tip şir. Folosind funcţia de

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

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

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

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

Folosind funcțiile *gets( )* şi *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 şiruri.

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

Un masiv de şiruri este un masiv bidimensional tip char compus din linii şi coloane. Pe fiecare linie din aşa masiv va fi înscris cîte un şir de caractere. Numărul maximal de şiruri 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 şir de caractere, fiecare avînd lungimea maximală de 34 de caractere. Pentru a accesa un şir din aşa masiv se va folosi sintaxa: propoziție[i], unde i este numărul rîndului din masiv unde se va afla şirul. Pentru a accesa un caracter al şirului i din masiv se va folosi sintaxa: *propoziție[i][j]* unde j este poziția caracterului în şirul i.

9. Structuri în C/C++.

Pînă în momentul de față au fost studiate tipurile de date compuse, elementele cărora aparțin aceluiaşi tip de date(simplu sau compus). În cazul unui masiv, elementele acestuia erau de acelaşi 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 şi păstrarea unei informații mai complexe, aşa că orarul lecțiilor, reuşita unui student etc. Dacă precăutăm cazul cu reuşita 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 aceleiaşi persoane. Ca urmare ar fi justificată tratarea lor ca o singură valoare compusă. Însă tipurile datelor deferă între ele: numele şi grupa vor fi de tip şir, notele la examen - de tip întreg, iar balul mediu calculat - de tip real(float). Gruprea acestor componente întro 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 şi 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 numeşte 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 şi 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 şablon. 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 şi trebuie declarate ca fiind de tipul structurii respective. Şablonul 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 reuşita 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 aşa 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 şablonul 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 reuşita unui student trebuie declarată o variabilă: *struct stud a;* Acum avem variabila *a* compusă din 5 cîmpuri, pentru care a fost rezervată memorie.

Dacă în program trebuie folosite câteva variabile de unul şi acelaşi 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 şi simbol " ; " la sfârşitul declarării structurii. Exemplu:

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

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

9.2. Inițierea variabilelor tip înregistrare.

În cazul, cînd valorile inițiale ale componentelor unei variabile-structuă 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* şi concomitent declarată variabila *a* cu iniţializarea valorilor pentru componentele sale.

O altă variantă de iniţializare a componentelor structurii este iniţializarea 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 funcţia principală *main(),* şi este locală dacă e declarată înăuntru funcţiei *main(),* sau înăuntru altei funcţii. Însă dacă e necesară iniţializarea unei structuri ce conţine şiruri, ea trebuie declarată înainte de funcţia *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 cîteva elemente fiecare avînd valoare de tip diferit. Însă adresarea către valorile ce se conţin în cîmpurile structurii este imposibilă folosind nemijlocit numele cîmpului 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 şi numele cîmpului respectiv folosind următoarea sintaxă: *nume_var.nume_cîmp* , unde *nume_var* este numele variabilei tip înregistrare, iar *nume_cîmp* – numele cîmpului respectiv din variabilă.

Notă: La prelucrarea valorilor din cîmpurile structurii se vor folosi toate funcţiile, instrucţiunile şi operaţiile aplicabile tipului cărui îi aparţin câmpurile structurii.

Exemplu: De alcătuit o structură care ar conţine informaţii despre un student (nume, grupa) şi 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 şi 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 cîteva elemente simple sau compuse, care la rîndul său pot deasemenea să conțină alte elemente. Acest fapt ne dă posibilitate să folosim unele structuri în calitate de cîmpuri 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 cînd structura A contine în componența sa un cîmp, care la rîndul său este deasemenea o structură B, atunci structura A trebuie declarată numai după ce va fi declarată structura B. Următorul exemplu foloseşte variabila *a* de tip structură *stud* imbricată, care conține informația despre un student şi rezultatele unei sesiuni. Informația despre rezultatele sesiunii este grupată într-o structură aparte cu numele *sesia* şi e folosită în calitate de cîmp *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 şi 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 reieşind din rezultatele la 3 examene. Din acest exemplu este simplu de observat sintaxa de adresare la o valoare dintr-un cîmp a unei structuri imbricate: se folosesc numele fiecărui cîmp imbricat, separat prin punct şi scrise în ordine descrescătoare pînă la cîmpul destinație. Sintaxa *a.nota.med* înseamnă ca se face o adresare la cîmpul *med* ce aparține unei structuri *(sesia)*, care la rîndul său este cîmp *(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 şi prelucrarea datelor ce se vor conține în cîmpurile structurii

conform şablonului numai pentru o înscriere: un student, o persoană ş.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 şi numărul maximal persoane înregistrate in aşa 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 aşa fel au fost rezervate 10 regiuni de memorie, fiecare avînd volumul necesar pentru a păstra o structură întreagă pentru prelucrarea informației despre reuşita unui student.

Adresarea in program la un element al masivului de structuri va fi făcută ca şi în cazul unui masiv simplu: se va indica numele masivului cu numărul de ordine al elementului in paranteze pătrate. Exemplu: *x[3]*. Însă aşa 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 acelaş timp cîmp al masivului va fi posibilă folosind sintaxa: *nume_msiv[k].Nume_cîmp*, unde *k* – este numărul de ordine a înregistârii necesare.
Exemplu: Pentru a înscrie în câmpul *"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 find 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 reuşita 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 tuturor posibilităților şi instrumentelor limbajului în care are loc programarea. În timpul studierii instrucțiunilor şi tipurilor noi în C/C++ apar posibilități de rezolvare eficientă a unor probleme complicate. Însă odată cu creşterea complicității problemei de rezolvat, se măreşte volumul programului şi 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 şi a treia - pentru afişarea rezultatelor. Folosind aceste funcții, dacă în program trebuie de îndeplinit o sarcină oarecare, atunci se apelează la funcția

respectivă asigurînd-o cu informaţia necesară pentru prelucrare. Folosirea funcţiilor în C++ presupune respectarea următoarelor concepţii de bază:

a) Funcţiile grupează setul de operatori pentru îndeplinirea unei sarcini concrete.
b) Programul principal apelează la funcţie, adresîndu-se la numele ei, după care urmează paranteze rotunde. Exemplu: *afişare()*.
c) După terminarea prelucrării informaţiei, majoritatea funcţiilor întorc programului principal valori de tipuri concrete. De exemplu: *int* sau *float,* care pot fi folosite în calcule.
d) Programul principal transmite funcţiilor parametrii (informaţia iniţială), inclusă în paranteze rotunde, care urmează după numele funcţiei.
e) Limbajul C++ foloseşte prototipi de funcţie pentru determinarea tipului valorii returnate de către funcţie, deasemenea a cantităţii si tipurilor parametrilor transmişi funcţiei.

Odată cu mărirea volumului şi complicităţii programului, folosirea funcţiilor va deveni condiţia principala pentru rezolvarea eficientă şi corectă. În acelaşi moment crearea şi folosirea funcţiilor sînt proceduri simple.

În timpul creării programului e necesar de rezervat fiecare funcţie pentru rezolvarea unei sarcini. Dacă apar situaţii, cînd funcţia rezolvă cîteva sarcini, ea trebuie divizată în cîteva funcţii mai mici. Fiecare funcţie creată trebuie să primească un nume unical. Ca şi în cazul cu variabilele, numele unei funcţii este un identificator şi e de dorit să corespundă cu sensul logic al sarcinei pe care o îndeplineşte.

Funcţiile în C++ se aseamănă la structură cu funcţia principală *main()*. În faţa numelui funcţiei se indică tipul ei, iar după numele funcţiei urmează lista de parametri descrişi înăuntrul parantezelor rotunde. Corpul funcţiei compus din operatori este amplasat după descrierea parametrilor şi-i înconjurat cu acolade deschisă şi închisă. Sintaxa descrierii unei funcţii este următoarea:

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

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

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

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

numele afişare indică (compilatorului C++ şi programatorului ce citeşte acest cod) că funcţia nu foloseşte parametri (informaţie iniţială pentru îndeplinirea sarcinei).

Următorul program foloseşte funcţia *afişare()* pentru afişarea mesajului pe ecran:

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

Ca şi orice program în C++ acest exemplu va fi îndeplinit începînd de la funcţia principală *main( )*. Înăuntru programului operatorul de apel la funcţie apelează la funcţia de afişare astfel: *afişare();* unde parantezele rotunde după identificator indică compilatorului că în program se foloseşte funcţia *afişare()*. Cînd programul va întîlni apelul la funcţie, se va începe îndeplinirea tuturor operatorilor din corpul funcţiei, şi după aceasta executarea programului principal va fi continuată cu operatorul amplasat nemijlocit după operatorul de apel la funcţie. Următorul program-exemplu conţine două funcţii, prima afişază pe ecran o salutare, iar a doua tema lecţiei:

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

În rezultatul îndeplinirii acestui exemplu vor fi afişate pe ecran două mesaje: *"Hello"* şi *"Funcţii în C++ ",* în ordinea cum apar în program.

Funcţiile prezentate mai sus îndeplinesc sarcini foarte simple. În aceste cazuri programul putea fi alcătuit fără folosirea funcţiilor, cu includerea aceloraşi operatori în corpul funcţiei *main()*. Însă funcţiile au fost folosite pentru a analiza descrierea şi folosirea ulterioară a funcţiilor. În timpul rezolvării problemelor complicate va fi posibil, în aşa fel, de simplificat rezolvarea, împărţind sarcina în module aparte realizate de funcţii. În acest caz va fi simplu de observat că analiza şi modificarea

funcţiilor este cu mult mai simplă decît prelucrarea unui program voluminos şi complicat. Mai mult ca atît, funcţia creată pentru un program poate fi folosită şi în alt program fără schimbări. În aşa mod pot fi alcătuite biblioteci de funcţii, folosirea cărora cu mult va micşora timpul folosit pentru alcătuirea programelor.

10.1. Transmiterea parametrilor în funcţie.

Pentru a mări posibilităţile funcţiilor din program, limbajul C++ permite de a transmite informaţie în ele. Informaţia iniţială transmisă din program în funcţie la momentul apelului acesteia se numeşte parametru. Dacă funcţia foloseşte parametri, ei trebuie descrişi în timpul descrierii funcţiei. În timpul descrierii parametrelor funcţiei se indică numele şi tipul fiecărui parametru în următorul mod:

tip_parametru nume_parametru;

Dacă funcţia conţine cîţiva parametri, ei vor fi descrişi împreună între parantezele rotunde după numele funcţiei despărţiţi prin virgulă după cum urmează:

```
tip_funcţie nume_funcţie (tip_parametru1 nume_parametru1,
                          tip_parametru2  nume_parametru2
.....
tip_parametruN  nume_parametruN );
```

Funcţia din exemplu următor foloseşte un parametru de tip întreg, care în rezultatul îndeplinirii sale, îl afişează 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, funcţia *număr()* este apelată de 3 ori cu diferite valori a parametrului. De fiecare dată cînd în program este întîlnită funcţia *număr()* valoarea parametrului este înlocuită în funcţie şi rezultatul est diferit.

Parametrii funcţiei 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 foloseşte funcția *muncitor();* care afişează numele şi 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 şi sînt afişate 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 şi cărora le se atribuie valorile parametrilor actuali se numesc parametri formali.

Aşadar în exemplul precedent valorile „*Ivanov"* şi *355.35* sînt parametri actuali, iar *char num[15]* şi *float* salariu sînt parametri formali.

Urmatorul exemplu foloseşte 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 foloseşte parametri, ea trebuie să indice numele unical şi tipul fiecărui parametru.
- Când programul apelează la funcție, compilatorul atribuie valoarea parametrlor de la stânga la dreapta.

- Valorile transmise din program în funcție, trebuie să coincidă ca număr, loc şi 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 cînd 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 numeşte şi tipul funcției.

Funcția din următorul exemplu adună 2 numere întregi şi 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 (viod) { 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. Cînd compilatorul întîlneşte operatorul *return*, el întoarce valoarea dată şi 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 ignoraţi, funcția terminîndu-se odată cu îndeplinirea operatorului *return.*

Funcția *suma()* din exemplu precedent este alcătuită din 3 instrucțiuni. Există posibilitatea de a micşora 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 calculă suma a două numere. În general, parametrul *return* foloseşte în calitate de parametrul o expresie ce are rezultate tip identic cu tipul funcției, adică tipul valorii returnate de funcție.

Nu toate funcţiile returnează valori de tip întreg. Următorul exemplu foloseşte funcţia *media()* , care returnează media aritmetică a 3 numere întregi. Evident rezultatul funcţiei poate fi şi 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 funcţie a fost folosită conversia *(float(a+b+c))* a uni 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 funcţiei *media()* în momentul descrierii acestea, indică tipul valorii returnate.

Există situaţii când operatorul *return* este folosit în funcţii 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 funcţie poate fi folosită în orice loc al programului, unde e posibilă folosirea unei valori de tip identic cu valoarea returnată. Cînd funcţia întoarce o valoare, această valoare poate fi atribuită unei variabile de acelaşi tip folosind operatorul de atribuire. Exemplu: *R=media (5,7,10);*

În continuare la acele spuse, însăşi numele funcţiei poate fi folosit, în cazul cînd există necesitatea folosirii valorii returnate de funcţie, de exemplu valoarea returnată poate fi folosită în interiorul funcţiei de afişare:

```
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 şi tipul parametrilor folosiți de funcție. În fiecare exemplu precăutat pînă în prezent descrieria funcției era făcută înaintea apelului ei din programul principal. Însă sînt situații, cînd unele funcții în program sînt apelate reciproc. În aşa cazuri este posibilă situația cînd o funcție va fi apelată înainte descrierii sale.

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

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

void afişare (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 îşi are prototipul său care descrie tipul valorii returnate, cantitatea şi tipul parametrilor. Este important de nu uitat de simbolul *";"* la sfârşitul fiecărui prototip. Lispa 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 şi domeniul de vizibilitate.

Funcţiile folosite în exemplele precedente îndeplinesc nişte sarcini relativ simple. La momentul, cînd funcţiile vor trebui să îndeplinească nişte sarcini mai complicate, va apărea necesitatea folosirii în funcţii a variabilelor proprii.

Variabilele declarate în cadrul funcţiei se numesc locale. Numele şi valoarea unei variabile locale sînt cunoscute numai funcţiei în care ea a fost declarată. Chiar faptul că variabila locală există este cunoscut numai funcţiei în care ea a fost declarată. Declararea variabilelor are loc la începutul funcţiei, îndată după acolada ce deschide corpul acesteia. Numele variabilei locale trebuie să fie unical numai în funcţia în care a fost declarată. O variabilă se numeşte locală, din cauză că este văzută numai din funcţia 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 funcţiei; *nume_f* - numele funcţiei; *tip_vl* - tipul variabilei;

numele_vl - numele variabilei**;**

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

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

Următorul exemplu foloseşte funcţia *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()* foloseşte 2 variabile locale *i* şi *k* de tip întreg. Variabila *k* se inițializează cu valoarea 1 chiar în momentul declarării sale. Anume 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 întîmplătoare. Anume 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 sfîrşit valoarea finală a lui *k* va fi întoarsă în program cu ajutorul operatorului *return ()*.

În timpul declarări 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. Aşadar, dacă două funcții folosesc acelaşi 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 foloseşte funcția *suma()* pentru adunarea a două numere întregi. Această funcție atribuie rezultatul îndeplinirii sale variabilei locale *x*. Însă funcția *main()* foloseşte în calitate de parametru transmis funcției *suma()* deasemenea o variabilă cu numele *x*. Din cele spuse mai sus rezultă că ambele variabile vor fi precăutate ca locale şi 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 foloseşte funcţia *suma()* pentru a calcula suma elementelor unui masiv unidimensional de tip întreg. În calitate de parametri se folosesc masivul şi 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 şi valoarea căreia sînt cunoscute pe parcursul întregului program. Despre existenţa unei variabile globale într-un program C++ ştie orice funcţie din acest program. Pentru a crea o variabilă globală se foloseşte declararea ei la începutul programului în afara oricărei funcţii. Orice funcţie, care va urma după aşa 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 funcţii din program, dar şi poate fi schimbată din oricare din funcţiile prezente în program.

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

```
# 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 prezenţa variabilelor globale în program adaogă noi posibilităţi, e de dorit de a evita folosirea lor frecventă. Din cauza, că orice funcţie din program poate schimba valoarea variabilei globale este foarte greu de urmărit toate funcţiile, care ar putea schimba această valoare, ceia ce duce la control dificil asupra execuţiei programului. Pentru a rezolva această problemă se poate declara variabila în corpul funcţiei *main()* şi apoi de a o transmite altor funcţii, în calitate de parametru. În acest caz în stivă va fi amplasată copia temporară a acestei variabile, iar valoarea iniţială (originalul) va rămâne neschimbată.

10.6. Conflicte dintre variabile locale şi globale.

În cazul cînd un program trebuie să folosească o variabilă globală, poate apărea situaţie de conflict, dintre numele variabilei globale şi numele variabilei locale. În aşa cazuri limbajul C++ oferă prioritate variabilei locale. Adică, dacă există variabilă globală cu acelaşi nume ca şi variabila locală, compilatorul consideră, că orice apel a variabilei cu aşa nume este un apel al variabilei locale. Însă apar situaţii, 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 foloseşte variabila globală *num*. Pe lângă această funcţia *afişare()* foloseşte variabila locală *num*. Folosind operatorul global de acces *(::)* funcţia 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++ conţine în cele mai frecvente cazuri o mulţime de variabile împreună cu descrierea lor. Fie variabila x descrisă în program prin declararea de forma *int x*; În acest caz compilatorul rezervează un spaţiu de 2 octeţi în memoria calculatorului. La rândul său, celula de memorie unde va fi alocată valoare variabilei x, îşi are adresa sa. De exemplu: 21650. Cînd variabila x primeşte o valoare, această valoare neapărat se înscrie în celula de memorie rezervată variabilei x. Deoarece, în cazul nostru variabila *x* îşi păstrează valorile pe adresa 21650, celula cu această adresă va conţine valoarea atribuită variabilei *x*.

În aşa 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++ conţine tehnici mai avansate de lucru cu valoarea unei variabile şi adresa de memorie unde se păstrează această valoare. În aceste scopuri se folosesc indicatori. Indicatorul reprezintă o variabilă, ce conţine 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 conţine 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 indicatiori 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 şi flotante. Există indicatori şi 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, lipseşte 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 îşi 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: presupunînd 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 şi valoarea sa nu se schimbă în timpul execuției procramului. 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 uşor se observă că adresa celulei de memorie se reprezintă printr-o valoare de tip întreg. În acelaşi timp această valoare nu poate fi schimbată în program şi expresia *&x=55;* este incorectă.

Analizînd toate aceste noțiuni, apare întrebarea: "Cu ce scop se folosesc indicatori, dacă valoarea variabilei şi 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* şi **q*. Astericsul * în acest caz indică că se apelează la conţinutul 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 aşa mod:

1) Variabila *q* poate primi valori numai în formă de adresă *q=&x* şi atribuirea de forma *q=21650;* este incorectă din cauza că aici se încearcă atribuirea unei valori întregi unui indicator şi 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-şi 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 afişate 2 expresii: *x=5* şi *x=6*. Prima valoare primită de variabila *x* este valoarea 5 atribuită la momentul iniţializării 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 funcţii, la crearea unui fişier, etc.

Este de remarcat faptul, că în anumite cazuri indicatorul neiniţializat reprezintă un pericol în program. De exemplu, dacă există necesitatea de a transmite datele pe adresa, ce conţine indicatorul, în condiţiile cînd acesta nu e corect iniţializat, se poate de şters informaţia importantă pentru funcţionarea sistemului operaţional, sau altor programe, fiindcă indicatorul neiniţializat poate indica la orice adresă de memorie inclusiv la adresa unde sînt păstrate date importante ale sistemului de operare. Un exemplu de iniţializare a indicatorului este: *int *q=&x.*

Cu scopul folosirii eficiente a memoriei limbajul C/C++ oferă programatorilor posibilitatea eliberării memoriei alocate, dacă e necesar. Pentru aceasta se foloseşte funcţia *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 acelaşi, dacă programul va fi executat pe orice calculator. Ultimele 2 mesaje vor fi diferite de la un calculator la altul în dependenţă de alocarea memoriei. Aici după folosirea funcţiei 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 şi funcţii.

Din temele precedente se ştie că în timpul lucrului cu o funcţie, la apelul ei se poate transmite atîţi parametri, de cîţi este nevoie în program. În acelaşi timp cu ajutorul operatorului *return*, se poate întoarce numai o singură valoare din funcţie. Şi situaţiile, cînd funcţia trebuie să calculeze şi să întoarcă cîteva valori în program par a fi nesoluţionate. În calitate de alternativă se pot folosi variabile globale, însă în acest caz deseori se pierde controlul asupra îndeplinirii programului şi se complică căutarea greşelilor.

În cazul, cînd există necesitatea de a întoarce din funcţie mai multe valori, este util de folosit indicatori. Fără folosirea indicatorilor, noi trebuie să transmitem parametri funcţiei după valoare. Aceasta înseamnă, că valoarea parametrului actual se atribuie parametrului formal. Aceşti parametri ocupă zone de memorie diferite, şi deci schimbarea valorii parametrului formal din funcţie 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 aceiaşi celulă de memorie. În aşa fel, dacă este schimbată valoarea

parametrului formal din funcție prin indicator, deasemenea este schimbată şi 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* şi *y=8*. Funcția *schimb()* schimbă valorile lor înmulțindu-le la 2. Însă, dacă valoarea lui *x* este schimbată în funcție şi 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 îşi păstrează valoarea. În aşa 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 şir de caractere. Şirul 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 şir de caractere. Exemplul următor calculează de cîte ori într-o frază dată se întîlneşte caracterul *'a'*:

```
# include <stdio.h>
int litera (char *s){
int k=0;
wutile (*s!='\0') {
prinf ("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 conţine %d caractere a”,p); }

Aici, la apelul funcţiei *litera()* în calitate de parametru se transmite nu adresa şirului de caractere *x*, ci însăşi şirul. Aceasta e condiţionat 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 conţine adresa elementelor masivului, iar **s* – valorile lor. În aşa mod, la prima iteraţie a ciclului *while* variabila **s* va avea valoarea egală cu primul caracter din şirul *x* transmis funcţiei. După îndeplinirea primei iteraţii funcţia măreşte valoarea variabilei *s* cu 1 unitate. Aceasta înseamnă că valoarea adresei primului caracter din şir va fi incrementată, în rezultat primindu-se adresa caracterului al doilea din şir. Ciclul se va termina cînd va fi depistat caracterul nul în şir.

Acelaşi lucru se poate de făcut şi 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 conţine %d  zerouri”,p);
getch(); }
```

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

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

programele unde se prelucrează cîteva masive. Următorul exemplu ilustrează acest fapt şi face posibil prelucrarea a 2 masive: *x* şi *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 ("Afişare masiv x:\n");
for (i=0; i<n; i++)  printf ("x[%d]=%d\n",i,x[i]);
printf ("Afişare masiv y:\n");
for (i=0;i <n; i++)  printf ("y[%d]= %d\n",i, y[i]);
getch();}
```

În exemplu precedent elementele masivul şi mărimea lui sînt întroduse în funcție, iar programul principal afişază rezultatul executării funcției. În calitate de parametri funcția *masiv()* foloseşte 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 aşa fel primindu-se adresa următorului element al masivului ce se prelucrează. Funcția întoarce în program valoarea mărimii masivului curent ce se atribuie variabilelor *n1* pentru masivul *x* şi *n2* pentru masivul *y*.

Avînd la dispoziție un aşa algoritm este uşor de compus programe ce prelucrează mai multe masive şi 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 şi 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 rînd din masivul bidimensional. Iar fiecare indicator (k[i]+j) în ciclu va duce la poziționare pe elementul coloanei j din rîndul i al masivului bidimensional.

12. Fişiere în C/C++.

În timpul executării programelor, în majoritatea cazurilor, există necesitatea afişării rezultatelor. Însă posibilitățile simplei afişări a datelor la monitor sînt destul de mici. Chiar şi folosirea opririi temporare a execuției programului, cu scopul de a da posibilitate utilizatorului să citească toată informația afişată, nu rezolvă pînă la sfîrşit problema. Dacă mesajul afişat dispare din limitele monitorului, afişarea 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 întrodusă se pierde.

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

Înscrierea sau citirea informației din fişier este efectuată cu ajutorul indicatorului la fişier. În timpul înscrierii sau citirii informației din fişier compilatorul foloseşte un nivel intermediar de legătură între program şi discul rigid, unde este păstrat fişierul. 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 fişier.

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

Sintaxa de declarare a unui indicator de fişier este următoarea:

*FILE *file_pointer;*

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

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

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

Prelucrarea fişierelor în limbajul C/C++ presupune îndeplinirea următorilor paşi:

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

12.1. Deschiderea fişierelor.

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

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

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

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

Există 3 modalități generale de deschidere a fişierului:

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

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

```
FILE *fişier;
fişier=fopen("info.txt", w");
```

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

```
FILE*fişier;
fişier=fopen("info.txt","r");
```

Pentru a imprima informaţia din fişier pe hîrtie cu ajutorul imprimantei, numele fişierului va fi *PRN*, iar modul de acces *"w":*

*FILE *fişier;*

fişier=fopen("PRN","w");

E de menţionat faptul, că şi numele fişierului şi caracterul ce determină modul de acces la fişier sînt delimitate de ghilimele duble. Aceasta este condiţionat de faptul, că parametrii în funcţia *fopen()* se transmit în calitate de şiruri de caractere. Folosind aceste posibilităţi, se poate culege de la tastatură numele fişierului dorit de utilizator:

char name [12];

*FILE *f;*

printf ("Culege numele fişierului\n");

gets (name);

f=fopen (name,"w");

În timpul lucrului cu fişierela în C/C++ este folosit un indicator special, în care se păstrează informaţia despre poziţia curentă de citire din fişier. În timpul citirii a datelor din fişier indicatorul determină porţia următoare de date care trebuie citită de pe disc.

Dacă fişierul e deschis pentru prima dată cu regimul de acces *"r",* indicatorul se amplasează pe primul simbol din fişier. În timpul îndeplinirii operaţiei următoare de citire, indicatorul se amplasează la începutul următoarei porţii de date. Mărimea pasului de citire din fişier depinde de cantitatea de informaţie citită din fişier. Dacă dintr-un pas se citeşte numai un caracter, atunci indicatorul se va amplasa pe caracterul următor, dacă se citeşte o structură, indicatorul va fi amplasat pe următoarea structură. La momentul, cînd toată informaţia a fost citită din fişier, indicatorul nimereşte pe un cod special numit sfîrşit de fişier: *(eof)*. Încercarea de citire din fişier după sfîrşitul fişierului duce le eroare.

Dacă fişierul este deschis cu regimul de acces *"w",* indicatorul deasemenea se amplasează la începutul fişierului în aşa fel primele date înscrise în fişier vor fi amplasate la începutul lui. În timpul închiderii fişierului, la sfîrşitul lui va fi înscris simbolul de sfîrşit de fişier *(eof)*. Dacă la momentul deschiderii în regimul de acces *"w"* fişierul există, toată informaţia deja existentă în el va fi nimicită şi deasupra ei va fi înscrisă informaţia nouă. Orice date precedente ce pot rămîne neşterse în fişier vor fi amplasate după codul sfîrşit de fişier *(eof),* deci accesul la ele va fi închis. În aşa fel toată informaţia din fişierul deschis în regim de acces *"w"* va fi nimicită, chiar şi în cazul cînd fişierul va fi deschis fără înscrierea în el a careva date.

Dacă fişierul se deschide în regim de acces *"a",* indicatorul se amplasează pe simbolul sfîrşit de fişier *(eof).* Orice informaţie în aşa fel înscrisă în fişier va fi amplasată după datele deja existente, iar după înscriere la sfîrşitul fişierului se adaogă codul sfîrşit de fişier *(eof).*

În unele cazuri apare situaţia cînd sistemul operaţional nu poate deschide fişierul indicat în funcţia *fopen().* Aceasta poate condiţionat de lipsa de loc pe discul rigid, sau de faptul că fişierul indicat pur şi simplu nu există. Este posibilă deasemenea situaţia cînd e necesară imprimarea datelor la imprimantă, iar aceasta nu-i inclusă sau lipseşte hîrtie. În cazul încercării folosirii fişierului ce nu poate fi deschis, programul va fi stopat în rezultatul unei greşeli de execuţie. Pentru a evita stoparea avariată a programului poate fi controlată starea de deschidere a fişierului cu ajutorul instrucţiunii condiţionate if, care va opri programul în caz de eşec. Aici va fi folosită particularitatea sistemului operaţional, care întoarce valoarea *NULL* în caz cînd apare greşală şi fişierul nu poate fi deschis. În acest caz codul *NULL* este întors în loc de adresa structurei-fişier şi programul se stopează. Condiţia pentru evitarea ieşirii avariate din program va avea următoarea sintaxă:

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

După ce toată informaţia este înscrisă în fişier sau citită din el e necesar de închis fişierul, adică de a întrerupe legătura dintre fişier şi program. Închiderea fişierului se realizează cu ajutorul funcţiei *fclose()* care are următoarea sintaxă:

fclose(f_pointer); unde *f_pointer* este numele indicatorului la fişier.

Odată cu închiderea fişierului, noi primim garanţia că toată informaţia din zona tampon într-adevăr a fost înscrisă în fişier. Dacă programul se termină pînă la închiderea fişierului, e posibilă situaţia cînd o parte din informaţie ce nu a fost înscrisă pe disc rămîne în zona tampon şi în rezultat este pierdută. În afară de aceasta dacă fişierul nu se închide corect, la sfîrşitul lui nu va fi înscris în modul necesar codul sfîrşit de fişier *(eof)* şi următoarea deschidere a fişierului va fi eronată. Deci sistemul operaţional va pierde accesul la fişier.

Mai mult ca atît, închiderea fişierului eliberează indicatorul şi după aceasta el poate fi folosit pentru accesul la alt fişier sau pentru îndeplinirea altor operaţii cu fişierul. Spre exemplu fie că trebuie de creat un fişier, de înscris informaţia în el, apoi de o citit din fişier. Menţionăm faptul, că după înscrierea datelor, fişierul trebuie închis şi numai după aceasta de-l deschis pentru citire, astfel avînd acces respectiv la datele din fişier:

```
#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 infirmaţiei 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 fişier este forma de bază pentru lucru cu fişiere. Cu toate că nu se bucură de o popularitate mare, aceste operații bine ilustrează principiile de bază în lucrul cu fişierele.

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

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

Următorul exemplu face posibilă înscrierea în fişier 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, fişierul este deschis în regimul de acces *"w"*. Şi dacă la momentul deschiderii fişierul 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()* şi înscrierea lor în fişier cu ajutorul funcției *fputc()* Este de menționat faptul, că cu acelaşi succes aici ar putea fi folosită funcția *putc()* cu aceiaşi parametri. Ciclul *do* va continua pînă cînd va fi detectată culegerea tastei *ENTER* şi după aceasta fişierul este închis.

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

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

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

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

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

Fiind deschis fişierul în regimul *"r",* este posibilă citirea informaţiei din el. Ciclul *while*, în care are loc citirea consecutivă a caracterelor din fişier este îndeplinit pînă cînd nu este detectat simbolul sfîrşit de fişier *EOF*, care este înscris la sfîrşitul oricărui fişier în momentul închiderii acestuia.

Funcţia de citire *fgetc(f1)* foloseşte alt indicator la acelaşi fişier – *f1*. Acest fapt este condiţionat de faptul, că chiar dacă la operaţiile de înscriere şi citire din fişier este folosit acelaşi nume de fişier, înscrierea sau citirea este recomandat de efectuat cu ajutorul diferitor indicatori.

12.2.2. Înscriere/citire de şiruri.

În cazul, cînd e necesar de înscris în fişier seturi de caractere, adică şiruri este binevenită funcţia *fputs();* care are următoarea sintaxă:

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

Funcţia *fputs()* efectuează înscrierea şirurilor în fişier sau imprimarea lor pe hîrtie fără înserarea caracterului *sfîrşit de linie*. Pentru ca fiecare şir înscris în aşa mod în fişier să înceapă din rînd nou, este necesară înserarea manuală a simbolului *sfîrşit de linie*.

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

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

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

Aici ciclul *while* va fi repetat pînă cînd va fi întrodus un şir de lungime 0. Funcţia *fputs()* va înscrie în fişier şirurile fiecare din rînd nou datorită inserării *fputs("\n",k);* La sfîrşitul prelucrării sale, fişierul *familia.txt* este neapărat închis fclose(k); Este de menţionat faptul, că pentru a imprima pe hîrtie familiile culese folosind imprimanta, este necesar de indicat numele fişierului *"prn"* în felul următor: *k=fopen ("prn","w");*. Pentru imprimarea corectă la imprimantă este necesară folosirea şirurilor cu lungime de 81 caractere, cu scopul ca şirul să încapă în întregime în lăţimea monitorului, înainte de a fi culeasă tasta ENTER.

Ciritea şirurilor de caractere din fişier este realizată de funcţia *fgets()* ce are următoarea sintaxă:

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

Următorul exemplu face posibilă citirea familiilor din fişierul "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("Informaţia cititădin fişier:/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 "sfîrşit de fişier". În cazul citirii informaţiei din fişier la nivel de şiruri de caractere, pentru a indica sfîrşitul fişierului se foloseşte codul *NULL,* iar codul *EOF* este folosit la citire caracterială. Funcţia *fgets()* va citi şirul în întregime pînă la codul „linie nouă", dacă lungimea lui nu depăşeşte valoarea *"l-1"* indicată în parametrii funcţiei.

Atrageţi atenţia la faptul, că funcţia *printf("%s",name);* nu foloseşte codul *"/n"* pentru deplasarea în rînd nou, fiindcă fiecare şir citit din fişier deacum conţine

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

12.2.3. Întrare/ieşire cu format.

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

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

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

Următorul exemplu demonstrează modul de înscriere în fişier a informației despre un set de produse. În fişier vor fi înscrise denumirile, prețurile şi 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 continuati? y/n\n");
răspuns=getch();}
fclose(f);}
```

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

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

monitor 2000 1

Atrageți atenția la simbolul *"\n"*- sfîrşit la linie la sfîrşitul şirului de control a formatului din funcția *fprintf(),* anume datorită lui, informația despre fiecare produs este înscris din rînd nou în fişier.

Citirea informației cu format din fişier se realizează cu funcția *fscanf(),* care are aceleaşi restricții ca şi funcția *scanf()*, şi foloseşte următoarea sintaxă:

fscanf (fp, format, data); care este identică cu sintaxa funcției *fprintf()* în ciea ce priveşte descrieria parametrilor.

În exemplu următor se realizează citirea informației din fişierul *"produs.txt"* despre produsele din depozit înscrise în fişier 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 fişier are loc chiar odată cu controlul condiției de continuare a ciclului *while*. Ciclu *while* va fi îndeplinit pînă cînd nu va fi detectat codul de sfîrşit de fişier care în acest caz este *EOF*.

Datele din fişier sînt citite în variabilele *name, preț* şi *unit* şi apoi afişate la monitor.

12.2.4. Fişiere şi structuri.

Pentru a înscrie o variabilă tip structură (înscriere) în fişier 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 foloseşte 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 fişier dintr-o încercare. Aici este recomandat de folosit valoarea 1. Valori mai mari ca 1 se folosesc în cazul înscrierii în fişier a unui masiv de structuri dintr-o singură încercare.
- *fp* este numele indicatorului la fişier.

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

În următorul exemplu este creat un masiv de structuri cu informția despre un grup de studenți ce conține numele studentului, anul naşterii, şi balul mediu. Întîi de toate masivul de structuri se completează cu informație, apoi structurile, cîte una cu ajutorul ciclului *for* şi funcției *fwrite()* sînt înscrise în fişier. Numele fişierului este numit de utilizator şi păstrat în variabila *"filename"* de tip şir de caractere:

```
#include<stdio.h>
#include<conio.h>
sctruct 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 studenţ \n");
scanf("%d"&n);
for (i=0; i<n; i++) {
printf ("culege numele studentului\n");
scanf ("%s", x[i] nume);
prinf ("culege anul naşteri \n");
scanf ("%d", x[i]. an);
printf (culege nota medie\n");
scanf("%f, &m);  x[i].med=m; }
printf ("culege numele fişierului \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 fişier cu numele, care a fost atribuit variabilei *filename* şi în el înscrise structurile cu informația despre studenți. Dacă vom deschide fişierul cu ajutorul unui redactor de texte obişnuit vom

observa în el un conţinut neînţeles. În realitate informaţia (sau mai bine zis structurile) ce se află în acest fişier este înţeleasă de compilator şi poate fi citită cu ajutorul funcţiei *fread()* ce are aceiaşi sintaxă ca şi funcţia *fwrite()*, în ceea ce priveşte descrierea parimetrilor:

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

În următorul exemplu se realizează citirea tuturor înscrierilor (structurilor) din fişierul creat în exemplul precedent şi afişarea lor la monitor. Este de menţionat momentul că funcţia *fread()*, în rezultatul executării sale întoarce o valoare ce corespunde cantităţii structurilor citite cu succes din fişier. În cazul nostru are loc citirea structurilor cîte una din fişier, deci funcţia în caz de succes va întoarce valoarea 1. În caz de creare sau de detectare a sfîrşitului de fişier funcţia va întoarce valoarea 0.

```
#include<stdio.h>
#include<conio.h>
sctruct 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 fişierului \n”);
scanf (“%s”, fn);
k=fopen (fn,”r”);
printf (“Informaţia citită din fişier:\n”);
while ( fread (&y[i], sizeof(y[i]), 1, k)==1) {
printf (“Numele studentului: %s\n”,y[i].nume);
printf (“Anul naşterii: %d\n”,y[i].an);
printf (“Media: %f \n”,y[i].med);  i++;}
fclose(k); getch();  }
```

Următorul tabel conţine descrierea tuturor posibilităţilor de intrare a datelor referitor la fişiere, inclusiv valorile întoarse de funcţii în caz de citire eronată:

Tipul de date	Funcţii de ieşire	Funcţii de intrare	Valoarea întoarsă la citire
caractere	*putc(); fputc();*	*getc(); fgetc();*	*EOF*
şiruri de caractere	*fputs();*	*fgets();*	*NULL*
date cu format	*fprintf();*	*fscanf();*	*EOF*
structuri	*fwrite();*	*fread();*	*0*

Anexa1. Funcţii de intrare-ieşire în C/C++

Funcţii de ieşire în C

Funcţiile de ieşire în orice limbaj de programare au destinaţia de a transfera datele din memoria calculatorului la alte dispozitive periferice de ieşire cum ar fi monitorul, imprimanta, un fişier pe disc etc. În timpul procedurii de ieşire se face o copie a datelor care vor fi trimise la dispozitivul de ieşire, păstrîndu-se originalul lor în memoria calculatorului. În limbajul de programare C există mai multe funcţii de ieşire. Şi folosirea unei sau altei funcţii depinde de tipul datelor ce vor fi extrase şi metoda de prezentare a lor. Cea mai simplă sintaxă o au funcţiile de extragere a caracterelor şi şirurilor de caractere.

Notă: Toate funcţiile de ieşire din limbajul C sîn susţinute şi în limbajul C++.

Funcţia *puts()*

Funcţia *puts()* are destinaţia de a afişa un şir de caractere la monitor. Sintaxa acestei funcţii este următoarea: *puts(P);* unde P este parametrul funcţiei şi poate fi:

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

Definiţie: Literal se numeşte un set concret de caractere care este inclus nemijlocit în instrucţiunile limbajului C/C++ în locul numelui constantei sau a variabilei.În cazul folosirii unui literal în calitate de parametru pentru funcţia puts() fraza ce trebuie afişată pe ecran este inclusă în parantezele rotunde şi delinitată de ghilimele. De exemplu: *puts("Hello world!");* Folosirea unei constante de tip şir în calitate de parametru al funcţiei puts() foloseşte 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 funcţiei *puts()* fară delimitarea cu ghilimele. A treia metodă de folosire a funcţiei *puts()* cu variabilă de tip şir în calitate de parametru este ilustrată în exemplul următor:

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

Folosirea constantelor şi variabilelor de alt tip în calitate de parametru al funcţiei *puts()* va duce la eroare de compilare. Unica diferenţa la folosirea literalelor şi constantelor sau variabilelor este faptul că literalul trebuie delimitat de ghilimele iar variabila şi constanta se folosesc fara ghilimele. Majoritatea compilatoarelor efectuiază trecerea în alt rind după îndeplinirea funcţiei *puts()*. Aceasta înseamnă, că după afişarea datelor pe ecran, cursorul în mod automat se deplasează la începutul următorului rind. 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 afişării 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 afişării unui singur simbol la ecran . Ca şi 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 afişarea numai a unui singur caracter şi instrucția de tipul *putchar('Da');* va duce la eroare de compilare.

Diferența principală la folosirea caracterelor şi şirurilor de caractere constă în faptul, că şirurile de caractere sînt delimitate de ghilimele, pe cînd caracterele se delimitate de apostrofe.

Majoritatea compilatoarelor nu efectuiază deplasarea cursorului în rind nou după executarea funcției *putchar()* şi el rămîne nemijlocit după simbolul afişat. Pentru trecerea în alt rind după afişare se recomandă folosirea simbolului *'\n'*. Unele compilatoare folosesc pentru afişarea caracterelor şi 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 descries 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 descries în biblioteca *conio.h* şi folosirea ei va fi însoțită de directiva *#include<conio.h>*.

Funcția *printf()*

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

#define MESAJ "Hello world!";

void main(void) { printf(MESAJ); printf("Bine ati venit"); }

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

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

printf("şirul cu format", lista datelor);

Primul parametru se numeşte şir de conducere sau şir cu format. Acest parametru se delimitează cu ghilimele şî indică compilatorului în ce poziție din şir trebuie să apară datele. Şirul cu format poate conține orice text împreună cu nişte etichete numite indicatori de format care determină tipul datelor şi amplasarea lor. Orice indicator de format începe cu simbolul procent (%), după care urmează un caracter ce indică tipul datelor. Aşa 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 afişat în format *%f* sau *%e* în dependență de faptul care formă de scriere este cea mai scurtă; *%c* – caracter; *%s* – şir de caractere.

În aşa 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 afişa pe ecran însăşi simbolul procent (%) este necesar de-l scris de 2 ori în felul următor: *printf("%%");* Litera 'd' indică compilatorului faptul că va trebui afişată 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 cărora este necesar de afişat pe ecran. Lista datelor se desparte de şirul cu forma prin virgulă. Toate elementele din lista datelor deasemenea se despart între ele prin virgulă. Cînd compilatorul prelucrează această funcție, el înlocuieşte 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)*. Şirul cu format poate conține nu numai indicatori de format, ci şi text obişnuit care conține şi indicatori de format. De exemplu: *printf("Este ora %d",5);* În rezultatul executării acestui exempplu pe ecran va fi afişat mesajul "*Este ora 5*". Acelaşi effect poate fi primit şi la folosirea funcției *puts("Este ora 5");* însă pentru combinarea textului cu valori, constante şi variabile numerice e necesară folosirea funcției *printf()*. Exemplu:

void main(void) {

int ora; ora=5; printf("Este ora %d",ora); }

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

void main(void) {

int ora,min; ora=5; min=25;

printf("Este ora %d şi %d minute",ora,min);}

În rezultat vom avea afişat la ecran mesajul: "*Este ora 5 şi 25 minute*". În cazul folosirii cîtorva parametri în lista datelor şi respectiv în şirul cu format,parametri din lista datelor trebuie să corespundă ca cantitate, poziție şi tip cu indicatorii din şirul 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 cîțiva parametri de acelaşi tip, ci şi 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 afişat mesajul: "*Prețul este de 5.210000 lei pentru 3 kg*." Aici întradevăr parametrii din lista datelor corespund ca cantitate, poziție şi tip cu indicatorii de format din şirul cu format.

Funcția *printf()* nu trece automat cursorul în alt rind după afişarea datelor. Adică dipă afişare cursorul rămîne în acelaşi rind nemijlocit după ultimul caracter afişat. Pentru trecerea cursorului î alt rind aici va fi folosită consecutivitatea de conducere *'\n'*, care trece cursorul pe prima poziție a rîndului următor ca şi în cazul funcției *puts()*. De exemplu: *printf("Prețul este de %f lei \npentru %d kg.",5.21,3);* Aici după afişare mesajul va vi amplasat în 2 rînduri:

Prețul este de 5.210000 lei

pentru 3 kg.

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

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

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

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

Aici numărul real va fi afişat cu 2 poziții după virgulă: *5.21*. Iar în versiunea *printf("Prețul este de %6.2f lei \npentru %d kg.",5.21,3);* vom avea _ _*5.21* unde '_' este spațiu. În general indicatorul de lățime a cîmpului pentru numerele reale are următoarea formă *'%k.rf*', unde *k* este numărul total de poziții (inclusive virgula) pe care va fi afişată cifra reală, *r* este numărul de poziții după virgulă. Dacă *k* va fi mai mare decît *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 şi o poziție este însăşi virgula. Dacă *k* va fi mai mic ca *r+2*, indicatorul de lățime a cîmpului pur şi simplu va fi ignorat. Indicatorul de lățime a cîmpului de forma *%6.4f* va prezenta cifra *5.21* în formă de *5.2100*. Pentru numerele întregi, caractere şi şiruri de caractere indicatorul de lățime a cîmpului are forma *%kd, %kc, %ks,* unde *k* este numărul total de poziții pe care va fi afişată valoarea. În acelaşi timp dacă *k* va fi mai mare ca lungimea reală a valorii afişate, înaintea acestei valori vor fi inserate spații în cantitate de *k-p*, unde *p* este numărul de poziții pe care real este amplasată valoarea afişată. Iar dacă *k* va fi mai mic ca lungimea reală a valorii, indicatorul de lățime a cîmpului pur şi 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.

Afişarea informației în C++

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

Pentru folosirea fluxului cout este necesar de inclus în textul programului C++ fişierul *<iostream.h>* cu ajutorul directivei *#include<iostresm.h>,* din cauza că el conţine descrierea fluxurilor de intrare şi ieşire în C++. Structura instrucţiunei ce foloseşte fluxul cout este următoarea: *"cout<<lista_de _date;"* unde simbolul '<<' este operatorul inserării şi indică compilatorului necesitatea afişării listei de date ce urmează după el. În calitate de informaţia afişată cu ajutorul fluxului cout pot servi literali, nume de constante şi variabile de orice tip. Mai mult ca atît, folosind unul şi acelaşi flux de ieşire se poate afişa pe ecran cîţiva argumenţi, cu condiţia că ei vor fi delimitaţi 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 şi funcţia *printf()* fluxul *cout* nu trece cursorul în rind nou automat după afişarea datelor. Pentru prezentarea comodă a datelor aici este necesară folosirea consecutivităţii de conducere *'\n'* după cum este arătat în exemplul precedent.

Funcţii de intrare în C.

Procesul de intrare a datelor presupune întroducerea informaţiei necesare pentru lucrul normal al programului. Informaţia întrodusă se amplasează în variabile, aceasta înseamnă că valorile venite de la utilizator în urma afişării comentariului corespunzător pentru întroducere sînt atribuite ca valori variabilelor ce se păstrează în memorie. În general intrarea datelor poate fi efectuată de la diferite surse aşa ca tastatura, memoria, unitatea de disc rigid sau flexibil, scanner, CD-ROM ş.a.

Intrarea datelor este un process ce determină lucrul de mai departe al programului. Corectitudinea datelor de intrare influienţează direct asupra corectitudinii rezultatelor obţinute cu folosirea acestor date de intrare. Datele de intrare pot fi atribuite ca valori numai variabilelor şi 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 şi analizate numai posibilitățile de intrare de la tastatură, lăsînd 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 şi char la folosirea funcției *getchar(),* fapt condiționat de standardul K&R C. Daca variabila va fi declarată de tip int, funcția *getchar()* va primi-o în calitate de valoare de intrare şi 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 numale 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 foloseşte argument, din această cauză parantezele rotunde de după numele funcției rămîn goale. După ce utilizatorul culege o tastă de la tastatură, *getchar()* afişază simbolul întrodus pe ecran. În acest caz nu e necesară culegerea tastei *Enter* după îndeplinirea funcției, din cauza că *getchar()* face posibilă întroducerea numai unui singur simbol, după ce programul trece la îndeplinirea de mai departe. Valoarea întrodusă este atribuită variabilei îndată după ce a fost cules un caracter. În timpul folosirii funcției *getchar(),* în textul programului trebuie inclus fişierul *<stdio.h>,* care conține descrierea prototipului acestei funcții.

Unele compilatoare C şi C++ folosesc funcția *getch()* analogică cu funcția *getchar()*. Însă descrierea prototipului funcției *getch()* se află în fisierul *<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 şi analiza rezultatelor nemijlocit după sfîrşitul

executării programului. Funcțiile *getch()* şi *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, ajungînd la sfîrşitul programului şi întîlnind funcția *getch(),* compilatorul va opri executarea programului pînă cînd va fi culeasă tasta *Enter*.

Notă: Limbajul C/C++ conține funcția gets() destinată întroducerii de la tastatură a şirurilor de caractere. Sintaxa şi principiul de lucru al acestei funcții vor fi studiate la tema "Şiruri de caractere".

Funcția *scanf()*

Funcția de intrare *scanf()* face posibilă întroducerea în calculator a datelor de orice tip. Funcția scanează tastatura, determină tastele care au fost culese şi apoi interpretează informația culeasă cu ajutorul indicatorilor de format, ce se află în componența funcției. Ca şi în cazul funcției *printf()* funcția *scanf()* poate avea mai mulți argumenți făcînd posibilă întroducerea valorilor de tip întreg, caracterial şi şir de caractere în acelaşi timp, adică în cadrul unei funcții. Lista de parametri ai funcției *scanf()* este compus din două părți ca şi la funcția *printf()*. Prima parte este şirul cu format, iar a doua lista cu date. Şirul 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 întroduse. Sintaxa funcției *scanf()* este următoarea:

```
scanf("şirul cu format", lista datelor);
```

De exemplu:

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

Să analizăm funcția `scanf("%d%f",&a,&b);`

Folosind funcția *scanf()* în timpul cînd se întroduc 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 cînd funcția *scanf()* întîlneşte formatul variabile a, ea îl

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

Procedura de intrare în C++

Compilatoarele limbajului C++ susțin funcțiile de intrare *gets(), getchar()* şi *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 şi 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 şi 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 şi atribuite valori variabilelor *a* şi *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 şi 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 şi dă posibilitatea de a introduce uşor 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 cîteva sarcini mai mici dar mai simple. Însă limbajul C conține şi o mulțime de funcții standarde care uşurează cu mult programarea. Aceste funcții oferă o varietate de facilități. În

plus, fiecare programator îşi poate construi propria sa bibliotecă de funcţii care să înlocuiască sau să extindă colecţia de funcţii standard ale limbajului.

Ca atare limbajul de programare C conţine în arsenalul său de instrucţiuni foarte puţine 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 funcţii standarde, care uşurează cu mult rezolvarea unor situaţii destul de complicate. Unele din aşa funcţii matematice ale limbajului C întîlnite mai des în practica de programare vor fi studiate în prezentul paragraf. Menţionăm faptul, că aceste funcţii sînt valabile şi în limbajul C++ şi folosirea lor într-un program în C sau C++ necesită includerea fişierului mat.h cu ajutorul directivei *#include<math.h>*.

1. Funcţia 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. Funcţia 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 [tg(x)];
   Exemplu: vezi funcția cos(x).
5. Funcția acos(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 asin(x).
   Prototip: double asin(double x); long double asinl(long double x);
   Efect: Întoarce valoarea arcsinus a numărului x [arcsin(x)];
   Exemplu: vezi funcția acos(x).
7. Funcția atan(x).
   Prototip: double atan(double x); long double atanl(long double x);
   Efect: Întoarce valoarea arctangentei a numărului x [arctg(x)];
   Exemplu: vezi funcția acos(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 calculînd deasemenea şi valoarea lui y.

Exemplu: În calitate de necunoscută aici este folosită numai variabila x. Avînd un x=8, operația k=frexp(x,y) calculează cifra reală (k), care trebuie înmulțită la 2^y pentru a primi rezultatul egal cu x=8, determinîndu-l în acelaşi timp şi pe y (valoarea puterii la care va trebui ridicată cifra 2). În cazul nostru x=8 şi k=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 rotungeşte 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 rotungită 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 rotungeşte pe x în partea de jos.

    Exemplu: vezi funcția ceil(x).

18. Funcţia fmod(x,y).

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

Efect: Întoarce valoarea restului de la împărţirea 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 împărţirea “<<x<<" la "<<y<<” etse = “<<k;
getch(); }
```

19. Funcţia modf(x,y).

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

Efect: Întoarce partea fracţionară a numărului real x, păstrîndu-se în y şi valoarea părţii 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<<
”şi partea fracţionară = “<<k; getch(); }
```

20. Funcţia 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 şi 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 foloseşte 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. Funcţii folosite la prelucrarea şirurilor de caractere.

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

1. Funcţia strcat();
   Prototip: char *strcat (char *dest, const char *sursa);
   Efect: Adaogă şirul sursă la sfîrşitul şirului destinaţie
   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. Funcţia strcmp();
   Prototip: int strcmp (const char * S1, const char* S2);
   Efect: Compară două şiruri.
   Limbajul C nu permite compararea a două şiruri în forma: *if (S1==S2) ;*
   Aici compararea este făcută cu ajutorul funcţiei strcmp() care întoarce valoare nulă, în caz că şirurile sînt identice, sau o valoare diferită de zero, în caz că şirurile nu coincid. După executarea funcţiei 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 daca 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 şiruri 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 cartere în 2 variabile de tip şir 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 şirul a cu primul caracter din şirul 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 daca 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 şiruri de caractere fără a face diferență între caractere minuscule şi cele majuscule.

   Exemplu: Echivalent cu funcția strncmp().

6. Funcția strlen();

   Prototip: size_t strlen(const char *S);

   Efect: Determină lungimea şirului de caractere S.

   În majoritatea cazurilor lungimea şirului nu coincide cu lungimea masivului în care se află şirul. Adică lungimea masivului este mai mare. Funcția strlen() determină lungimea reală a şirului şi întoarce o valoare de tip întreg egală cu cantitatea de caractere ce se află în şir.

   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 şirul S2 în şirul S1. După îndeplinirea funcției *strcpy (S1,S2);* şirul S1 îşi va pierde valoarea inițială şi 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 şirul S1, care primul a fost întîlnit în şirul 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” şi fam=“Ivanov”, atunci rezultatul funcției *k=strcspn(name,fam);* va fi k=4, din cauză că pe locul patru în numele “Step**a**n” se află caracterul “a” care primul a fost depistat în “Iv**a**nov” .

9. Funcția strspn();

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

   Efect: Determină poziția caracterului din şirul S1 începînd 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 ("Începînd cu simbolul %c şirul %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 începînd cu caracterul cu numărul de ordine 2 variabila *nume* diferă de variabila *fam*.

10. Funcția strdup();

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

Efect: Dublează şirul de caractere S. În caz de succes funcţia strdup() întoarce ca valoare indicatorul adresei de memorie, ce conţine şirul dublat. Şi întoarce valoare nulă în caz de eroare. Funcţia strdup(S) face o copie a şirului S, obţinînd spaţiu prin apelul funcţiei malloc(). După folosirea şirului 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=strdup(a); puts (b); free (b);}
```

Aici *a* este un şir de caractere, iar *b* un indicator la adresa de mmorie unde se va înscrie şirul dublat. La executarea funcţiei *strdup()* valoarea şirului din *a* este copiat într-o locaţie de memorie, adresa căreia se află în *b*. Funcţia *puts()* afişază conţinutul şirului dublat la monitor. Funcţia *free()* eliberează memoria ocupată pînă acum de şirul dublat.

11. Funcţia strlwr();

Prototip: char *strlwr (char *S);

Efect: Trece toate caracterele din şirul S în echivalentul lor minuscul. În calitate de parametru funcţia foloseşte o variabilă de tip şir de caractere. În rezultatul executării acestei funcţii, dacă în şir se vor conţine caractere majuscule, ele vor fi transformate în echivalentul lor minuscul, iar dacă în şir caractere majuscule nu se vor conţine - şirul 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. Funcţia strupr();

Prototip: char *strupr(char *S);

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

Exemplu: Echivalent cu funcţia strlwr();

13. Funcția strncat()

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

Efect: funcția strncat() adaogă un număr egal cu *k* caratere de la începutul şirului sursa la sfărşitul şirului 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 şirul *b[10]*, (adica subşirul “Pas”) vor fi adaogate la sfîrşitul şirului *a[20]*. Rezultatul executării funcției strncat() va fi de tip şir de caractere şi se va conține în vaiabila *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 numar dat de caractere dintr-un şir 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 şirul sursă la începutul şirului destinație. Dacă şirul destinație va avea lungime mai mare ca N, atunci rezultatul va avea începutul egal cu caracterele copiate, iar sfîrşitul inițial. Valoarea rezultatului se va conține în şirul 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 şirul *S. În caz că n>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 funcţiei strnset() va fi întoarsă o valoare de tip şir care va fi înscrisă în şirul destinaţie . Din exemplu de mai sus *a =”WWWdent”*.

16. Funcţia strrev();

Prototip char *strrev(char *s);

Efect: Funcţia strrev() inversează şirul de caractere S. După execuţia funcţiei strrev() primul caracter din şir va fi schimbat cu locul cu ultimul caracter, caracterul 2 cu penultimul, etc. fără a lua în consideraţie caracterul nul.

Exemplu:

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

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

17. Funcţia strstr ();

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

Efect: Funcţia strstr() determină dacă şirul S2 se conţine în şirul S1. Funcţia întoarce un indicator la caracterul din S1 începînd cu care a fost depistat şirul 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: “subşirul este: national ;”. Dacă şirul 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 şirul 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 aşa caracter nu există în şirul 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);}
elseprintf("Caracterul n-a fost gasit\n");
getch();  }
```

În rezultatul îndeplinirii acestui exemplu la monitor va fi afişat mesajul: *Caracterul r are poziția 2 în şirul TurboC++* şi apoi datorită funcției *puts(ptr);* va fi afişat şirul 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 şi returnează un indicator la şirul 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 afişa: *Eroarea este: Invalid format.*

20. Funcția strpbrk();

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

Efect: Funcția caută în şirul S1 primul caracter ce există şi în şirul 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 în 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 şirul S. În caz de succes funcția întoarce un indicator la ultimul caracterdin 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 afişat mesajul: *Caracterul r este pe poziţia 8* şi apoi datorită funcţiei *puts(ptr);* va fi afişat şirul S trunchiat de la ultimul caracter *r* la care indică *ptr*: *„re”*.

22. Funcţia strset();

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

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

Rezultatul final se va păstra în acelaşi şir S.

Exemplu:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char s[10]=”student”;
printf(“Şirul iniţial - %s\n”,s);
strset(s,’a’);
printf(“Şirul final - %s\n”,s);
getch();}
```

În rezultat va fi primit: *Şirul iniţial – student, Şirul final – aaaaaaa.*