Lucrarea nr. 2

ELEMENTE GENERALE ALE LIMBAJULUI C

1. Scopul lucrării

Lucrarea are ca scop prezentarea elementelor de bază ale limbajului C.

2. Noțiuni teoretice

2.1 Structura programelor

Orice activitate de programare începe cu editarea programului, în conformitate cu regulile sintactice şi semantice aferente limbajului. Se elaborează astfel aşa-numitul ***„program sursă",*** care se păstrează într-un fişier (***fişier sursă***) sau mai multe. Aceste fişiere au extensia ***.c*** pentru limbajul C şi ***.cpp*** pentru limbajul C++.

Pentru a putea fi executat, programul trebuie să parcurgă o serie de etape:

- etapa de compilare care presupune rezolvarea directivelor către preprocesor şi transpunerea programului sursă în ***„program obiect"*** (prin compilarea unui fişier sursă se obține un ***fişier obiect*** cu extensia ***.obj***):

- etapa de link-editare care presupune legarea programului obiect obținut cu bibliotecile de sistem; rezultă un ***„program executabil"*** (în urma link-editării se obține un ***fişier executabil*** cu extensia ***.exe***)

- ***etapa de lansare*** în execuție.

Un program sursă este compus din una sau mai multe funcții dintre care una singură trebuie numită *main()*.

Exemplul următor conține o singură funcție, funcția *main()*. Către această funcție principală sistemul de operare transferă controlul la lansarea în execuție a programului.

Ex. 1

```
#include<stdio.h>
main()
{
printf ("Program in limbajul C!");
}
```

Dacă sunt parcurse în mod corect etapele de compilare şi link-editare ale programului, la lansarea lui în execuție va apare mesajul:

Program in limbajul C!

Acoladele { } încadrează o construcție (instrucțiune compusă sau bloc) alcătuită din declarații şi instrucțiuni aşa cum este cazul corpului unei funcții.

2.2 Variabile. Tipuri de variabile. Declarare

Limbajul C permite utilizarea de variabile (nume simbolice) pentru memorarea datelor, calculelor sau rezultatelor.

În cadrul programelor C este obligatorie declararea variabilelor, care constă în precizarea tipului variabilei. În urma declarației, variabila determină compilatorul să-i aloce un spațiu corespunzător de memorie.

În limbajul C există cinci tipuri de variabile:

- caracter: **char,**
- întreg: **int,**
- real în virgula mobilă în simplă precizie: **float,**
- real în virgula mobilă în dublă precizie: **double** şi
- tip de variabilă neprecizat sau inexistent: **void**

Primele 4 tipuri aritmetice de bază pot fi extinse cu ajutorul unor declarații suplimentare, cum ar fi:

- **signed** (cu semn)
- **unsigned** (fără semn)
- **long** (lung)
- **short** (scurt).

Un exemplu de set de tipuri de variabile obținut cu ajutorul acestor declarații este prezentat în Tab.1.

Exemple de declarații de variabile:

```
…int i, j, k;
double val, set;
unsigned int m;
```

Programul următor utilizează declarații multiple de variabile:

Ex. 2

```
#include<stdio.h>
main()
{
int nr;
char ev;
float timp;
nr=5;
ev = 'S';
timp = 11.30;
printf("Evenimentul %c are numărul  %d ", ev, nr);
printf("si a avut loc la %f", timp);
}
```

În urma execuției acestui program se va afişa:

Evenimentul S are numărul 5 şi a avut loc la 11.300000

Tab.1 Tipuri de variabile în limbajul C

Tip	Spațiul (biți)	Domeniul de valori
char	8	- 128÷127
unsigned char	8	0÷255
signed char	8	-128 ÷127
int	16	- 32768÷32767
unsigned int	16	0÷65535
signed int	16	- 32768 ÷ 32767
short int	16	- 32768÷32767
unsigned short int	16	0÷65535
signed short int	16	- 32768÷32767
long int	32	-2.147.483.648 ÷ 2.147.483.647
signed long int	32	-2.147.483.648 ÷ 2.147.483.617
unsigned long int	32	0÷4.294.967.295
float	32	10^{-37} ÷10^{37} (6 digiti precizie)
double	64	10^{-308} ÷10^{308} (10 digiti precizie)
long double	80	10^{-4932} ÷10^{4932} (15 digiti precizie)

2.3 Funcția printf( )

Funcția **printf()** permite afişarea textului aflat între ghilimele precum şi valori ale diferitelor variabile din program, utilizând anumite notații denumite ***specificatori de format*** care realizează conversia datelor într-o formă adecvată pentru afişare.

Prototipul funcției **printf()** se găseşte în fişierul antet **stdio.h** şi de aceea este nevoie de declarația preprocesor: **#include<stdio.h>** la începutul fiecărui program care foloseşte această funcție.

Formatele specifice utilizate de funcția **printf()** sunt:

%c — pentru afişarea unui singur caracter;
%s — pentru afişarea unui şir de caractere;
%d — pentru afişarea unui număr întreg (în baza zece) cu semn;
%i — pentru afişarea unui număr întreg (în baza zece) cu semn ;
%u — pentru afişarea unui număr întreg (în baza zece) fără semn;
%f — pentru afişarea unui număr real (notație zecimală);
%e — pentru afişarea unui număr real (notație exponențială);
%g — pentru afişarea unui număr real (cea mai scurtă reprezentare dintre **%f si %e);**
%x — pentru afişarea unui număr hexazecimal întreg fără semn;
%o — pentru afişarea unui număr octal întreg fără semn
%p — pentru afişarea unui pointer (a unei adrese);

În plus se pot folosi următoarele prefixe:

l cu d, i, u, x, o permite afişarea unei date de tip long
l cu f, e, g — permite afişarea unei date de tip double
h cu d, i, u, x, o — permite afişarea unei date de tip short
L cu f, e, g — permite afişarea unei date de tip long double.

Exemple:

%ld - permite afişarea unei date de tip long int
%hu - permite afişarea unei date de tip short unsigned int

În exemplul 2 afişarea valorii reale s-a realizat cu şase zecimale şi nu cu două conform operației de atribuire inițiale a acestei variabile. Pentru afişarea corectă, formatul de scriere **%f** trebuie însoțit de o notație suplimentară care specifică dimensiunea câmpului de afişare a datelor şi precizia de afişare a acestora. Această notație suplimentară se introduce între simbolul **%** şi simbolul **f** şi are forma generală

%-m.nf cu **m** şi **n** numere întregi având următoarele semnificații:

- semnul "**-**", în cazul în care este folosit, realizează alinierea la stânga a informației afişate în cadrul câmpului de afişare; în lipsa acestuia, implicit alinierea se face la dreapta.
- **m** precizează dimensiunea câmpului de afişare (numărul de coloane)
- **.n** reprezintă numărul de digiți din partea zecimală (precizia de afişare a numărului)

Astfel, în exemplul 2, dacă în locul specificației de format **%f** am fi trecut **%5.2f**, la consolă ar fi apărut mesajul:

Evenimentul S are numărul 5 şi a avut loc la 11.30

Ex. 3

```
#include<stdio.h>
main()
{
float valoare;
valoare = 20.13301;
printf ("%8.1f%8.1f\n", 22.5,425.7);
printf ("% -8.1f % -8.1f\n.", 22.5,425.7);
printf ("%f\n",valoare);
printf ("%5.2f\n", valoare);
printf (" %012f\n", valoare) ;
printf ("%10f\n", valoare);
}
```

În urma executării programului din ex.3, va rezulta:

				2	2	.	5				4	2	5	.	7
2	2	.	5					4	2	5	.	7			
2	0	.	1	3	3	0	1	1							
2	0	.	1	3											
	2	0	.	1	3										
0	0	0	2	0	.	1	3	3	0	1	1				
2	0	.	1	3	3	0	1	1							

Se observă că prezența unui zero înaintea numărului care specifică dimensiunea câmpului de scriere determină la afişare umplerea cu zero a spațiilor goale.

2.4. Secvențe de evitare (escape)

În exemplul 3, caracterul **"\n"** inserat în şirul de caractere din apelul funcției **printf()** a determinat afişarea pe o linie nouă (*carriage return-linefeed).* Acest caracter este un exemplu de secvență **escape,** numită aşa deoarece simbolul **(\) backslash** este considerat caracter escape, care determină o abatere de la interpretarea normală a şirului. Aceste secvențe escape sunt:

\a *beep* alarmă-sonoră;
\b *backspace* spațiu înapoi;
\f *formfeed* linie nouă, dar pe coloana următoare celei curente;
\n *newline* determină trecerea la linie nouă, pe prima coloană;
\r *carriage return* determină revenirea la începutul liniei;
\t *tab* determină saltul cursorului din 8 in 8 coloane;
\\ *backslash* ;
\' apostrof simplu;
\" ghilimele;
\0 *null* ;
\ddd valoare caracter în notație octală (fiecare d reprezintă un digit);
\xdd valoare caracter în notație hexazecimală;

2.5. Funcția *scanf()*

O altă funcție des utilizată a limbajului C este funcția de introducere a datelor ***scanf().*** În mod similar cu *printf()*, apelul funcției *scanf()* implică utilizarea unui şir de caractere de control urmate de o listă de argumente. Dar, în timp ce *printf*() utilizează nume de variabile, constante şi expresii, *scanf()* utilizează pointeri la variabile.

Exemplul 4 este o variantă a programului din exemplul 2 în care, în plus, se utilizează funcția *scanf()*.

Ex. 4

```
#include<stdio.h>
main()
{
int nr;
char ev;
float timp;
printf ("Introduceți pozitia, evenimentul, timp:");
scanf ("%c %d %f", &ev, &nr, &timp);
printf ("Evenimentul %c are numărul %d ", ev, nr);
printf (" şi a avut loc la %5.2f", timp);
}
```

Execuția programului poate fi:
Introduceți pozitia, evenimentul şi timpul: A 6 11.30
Evenimentul A are numărul 6 şi a avut loc la 11.30

Valorile variabilelor *ev, nr şi timp* au fost introduse de la tastatură. Pentru separarea lor a fost utilizat spațiu (blank). Putea fi folosit return sau tab; orice alt separator (linie, virgula) nu realizează această separare.

3. Problemă rezolvată

3.*1 Să se scrie un program care permite aflarea codului numeric al unei taste în zecimal, hexa si octal.*

```
#include<stdio.h>
#include<conio.h>
void main(void)
{char caracter;
clrscr();
printf("Acest program afiseaza codul unei taste in zecimal, hexa si octal.\n");
printf("Apasati o tasta:");
scanf("%c",&caracter);
printf("Codul tastei \"%c\" este %d (in decimal), %x (in hexa), %o (in octal)\n", caracter, caracter, caracter, caracter);
getch();
}
```

4. Chestiuni de studiat

4.1 Studierea noțiunilor teoretice şi a exemplelor prezentate.

4.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizați.

Lucrarea nr. 3

POINTERI ÎN LIMBAJUL C

1. Scopul lucrării

Lucrarea are ca scop prezentarea utilizării variabilelor pointer în limbajul C.

2. Noțiuni teoretice

2.1 Declararea variabilelor pointer

Variabilele pointer (indicator) reprezintă adrese ale unor zone de memorie. Pointerii sunt utilizați pentru a face referire la date cunoscute, prin adresele lor. Puterea şi flexibilitate în utilizarea pointerilor, specifică limbajului C, reprezintă un avantaj față de celelalte limbaje (de ex. Pascal).

Există 2 categorii de pointeri:

- pointeri de date (obiecte) care conțin adrese ale unor variabile sau constante din memorie;
- pointeri de funcții care conțin adresa codului executabil al unei funcții.

În plus, există şi pointeri de tip **void** (o a treia categorie), care pot conține adresa unui obiect oarecare.

Declararea unui pointer de date se face cu sintaxa:

tip *var_ptr;

Prezența caracterului * definește variabila **var_ptr** ca fiind de tip pointer, în timp ce **tip** este tipul obiectelor a căror adresă o va conține (numit şi tipul de bază al variabilei pointer **var_ptr**).

Pentru pointerii de tip **void** se foloseşte declarația:

void * v_ptr;

Exemplul următor conține un set de declarații de variabile pointer:

Ex.1:

```
int *iptr;
float *fptr, val;
void *v_adr;
int * tabptr [10];
float ** dptr;
```

În această secvență de program, variabila **iptr** este un pointer de obiecte **int**, iar variabila **fptr** un pointer de obiecte **float**. Tot aici, **tabptr** este un tablou de 10 pointeri de tip **int**, iar **dptr** va putea conține adresa unui pointer de obiecte **float** (se realizează o dublă indirectare, pointer la pointer).

Deoarece, la compilare sau în timpul execuției programului nu se fac verificări ale validității valorilor pointerilor, orice variabilă pointer trebuie inițializată cu o valoare validă, 0 sau adresa unui obiect sau a unei funcții, înainte de a fi utilizată.

Inițializarea cu valoarea 0 a unei variabile pointer indică faptul ca aceasta nu conține adresa unui obiect sau a unei funcții. Uzual, în aceste cazuri, se foloseşte pentru atribuire identificatorul **NULL(=0)**, care este declarat în fişierele antet (**stdio.h, stdlib.h** etc.).

Utilizarea variabilelor pointer implică folosirea a doi operatori unari: **operatorul &** ce permite aflarea adresei unei variabile oarecare şi **operatorul *** care permite accesul la variabila adresată de un pointer.

Astfel, în cazul unei variabile **var** de tipul **tip**, expresia:

&var se citeşte: **"adresa variabilei var"**

iar rezultatul este un pointer de obiecte **tip** şi are valoarea adresei obiectului **var**.

În cazul unei variabile pointer de obiecte **tip**, numită **ptr**, expresia:

***ptr** se citeşte: **"la adresa ptr"**

iar rezultatul este de tipul **tip** şi reprezintă obiectul adresat de variabila pointer **ptr**. Expresia ***ptr** poate fi utilizată atât pentru a aflarea valorii obiectului, dar şi în cadrul unei operații de atribuire.

Utilizarea acestor operatori este prezentată în exemplul următor, în care, pentru afişarea adreselor în hexazecimal se foloseşte funcţia **printf()** împreună cu specificatorul de format **%p**:

Ex.2:

```
#include <stdio.h>
void main(void)
{
int var=5, *ptr;
printf("\n Variabila var se află la adresa:%p", &var);
printf("\n şi are valoarea var=%d",var);
ptr=&var;
printf("\n Variabila ptr are valoarea:%p", ptr);
printf("\n şi conţine adresa obiectului: %d",*ptr);
*ptr=10;
printf("\nAcum, variabila var are valoarea %d\n",var);
}
```

În urma execuţiei programului se afişează:

```
Variabila var se află la adresa: 1A56
şi are valoarea var=5
Variabila ptr are valoarea: 1A56
şi conţine adresa obiectului: 5
Acum, variabila var are valoarea 10
```

În urma operaţiei de atribuire **ptr=&var**, variabila pointer **ptr** preia adresa variabilei **var**, astfel încât cele două obiecte ***iptr** şi **var** devin identice, reprezentând un întreg cu valoarea **5** de la adresa **1A56**. În aceste condiţii, expresia ***ptr** poate fi folosită în locul variabilei **var**, cu efecte identice. De aceea, atribuirea ***ptr=10** are ca efect modificarea valorii variabilei **var** din **5** în **10**.

2.2 Operaţii arimetice cu pointeri

Operaţiile aritmetice ce se pot efectua cu pointeri sunt: compararea, adunarea şi scăderea. Aceste operaţii sunt supuse unor reguli şi restricţii specifice. În cazul în care tipurile asociate operanzilor pointer nu sunt identice, pot apare erori, care nu sunt întotdeauna semnalate de compilator. În acest sens, se recomandă conversia de tip explicită cu operatorul **cast**, de forma **(tip*)**.

Operatorii relaţionali permit compararea valorilor a doi pointeri:

```
……..
int *ptr1,*ptr2;
if(ptr1<ptr2)
  printf("ptr1=%p <ptr2=%p",ptr1,ptr2);
……
```

În multe situaţii este necesară compararea unui pointer cu 0, pentru a verifica dacă adresează sau nu un obiect:

```
……..
if(ptr1==NULL)…/* ptr1 este un pointer nul*/
else…          /* ptr1 este un pointer nenul*/
……..
```

sau, sub forma:

```
……
if(!ptr1)…     /* ptr1 este un pointer nul*/
else …         /* ptr1 este un pointer nenul*/
……
```

Pot fi efectuate operaţii de adunare sau de scădere între un pointer de obiecte şi un întreg. Deoarece un pointer este o valoare care indică o anumită locaţie din memorie, dacă adăugăm numărul 1 acestei valori, pointerul va indica următoarea locaţie din memorie. Deci, în cadrul acestor operaţii intervine şi tipul variabilei.

Regula după care se efectuează aceste operaţii este următoarea:

în cazul unei variabile pointer **ptr**:

tip *ptr;

operaţiile aritmetice:

ptr+n şi ptr-n

corespund adăugării/scăderii la adresa **ptr** a valorii

n*sizeof(tip).

De exemplu:

```
……..
int *ip;              /*  sizeof(int)=2     */
float *fp;            /*  sizeof(float)=4   */
double *dp1, *dp2;    /* sizeof(double)=8 */
…….
dp2=dp1+5;   /*  dp2<-- adresa_dp1+5*8   */
fp=fp-2;     /*  fp<-- adresa_fp-2*4     */
ip++;        /*  ip<-- adresa_ip+1*2     */
dp1--;       /*  dp<-- adresa_dp-1*8     */
```

În acelaşi context, se poate efectua scăderea a doi pointeri de obiecte de acelaşi tip, având ca rezultat o valoare întreagă ce reprezintă raportul dintre diferenţa celor două adrese şi dimensiunea tipului de bază, ca în exemplul:

```
int i;
float *fp1,*fp2;   /*sizeof(float)=4*/
i=fp2-fp1;
/*i=(adresa_fp2-adresa_fp1)/sizeof(float)*/
```

Având în vedere importanța tipului pointerilor în cadrul operațiilor de adunare şi scădere, operanzii nu pot fi pointeri de funcții sau pointeri **void**.

2.3 Variabile dinamice

Pentru tipurile de date la care se cunoaşte dimensiunea zonei de memorie necesară, aceasta este fixată în urma declarației, înaintea lansării în execuție a programului.

În cazul structurilor de date a căror dimensiune nu este cunoscută sau se modifică în timpul execuției programului, este necesară o alocare prin program a memoriei, în timpul execuției. Memoria alocată este folosită, iar atunci când nu mai este utilă, se eliberează tot în timpul execuției programului.

Variabilele create astfel se numesc dinamice.

Prototipurile funcțiilor utilizate pentru alocarea şi eliberarea memoriei în timpul execuției programului se află în fişierele **alloc.h** şi **stdlib.h**.

O funcție des utilizată pentru alocarea dinamică a memoriei este **malloc()** şi are prototipul:

void*malloc(unsigned nr_octeți);

Prin parametrul funcției **malloc()** se precizează dimensiunea în octeți a zonei de memorie solicitate. Dacă operația de alocare reuşeşte, funcția întoarce un pointer care conține adresa primului octet al zonei de memorie alocate. În caz contrar (spațiul disponibil este insuficient) pointerul rezultat este **NULL** (=0).

Pentru o bună portabilitate a programelor, este recomandabil, în apelul funcției **malloc()**, să se utilizeze operatorii **cast** (pentru conversie de tip la atribuire) şi **sizeof** (pentru precizarea dimensiunii zonei solicitate).

De exemplu, dacă se doreşte alocarea unei zone de memorie pentru **10** valori de tipul **float**, se poate proceda astfel:

```
…….
  float *fp;
  fp=(float*)malloc(10*sizeof(float));
…….
```

Eliberarea memoriei (rezultate în urma unei alocări dinamice) se face atunci când variabila dinamică nu mai este utilă, iar spațiul alocat poate fi refolosit. În acest caz se poate folosi funcția **free()** care are prototipul:

void free(void*ptr);

Parametrul funcției **free()** este un pointer ce conține adresa zonei care trebuie eliberată şi care obligatoriu este rezultatul unui apel al unei funcții de alocare dinamică (de tip **malloc()**).

Astfel, zona alocată anterior poate fi eliberată prin apelul:

```
…….
  free(fp);
…….
```

Zona de memorie alocată astfel este echivalentă cu un tablou. Elementele acestuia pot fi referite indexat. De exemplu **fp[5]** este obiectul **float** de la adresa **fp+5**.

3. Problemă rezolvată

3.1 Acest program exemplifică regulile specifice operațiilor aritmetice cu pointeri.

```
#include <stdio.h>
void main(void)
{
int a=5,b=10, *iptr1, *iptr2,i;
float m=7.3, *fptr;
iptr1=&a;iptr2=&b;
fptr=&m;
printf("\n fptr=%u, *fptr=%2.1f, &fptr=%u", fptr,
*fptr, &fptr);
fptr++;printf("\n Incrementare fptr:");
printf("\n fptr=%u, *fptr=%2.1f, &fptr=%u", fptr,
*fptr, &fptr);
printf("\n iptr1=%u, *iptr1=%d, iptr2=%u,
*iptr2=%d", iptr1, *iptr1, iptr2,*iptr2);
i=iptr1-iptr2;
printf("\n Diferenta pointerilor iptr1 si iptr2
este=%d",i);
iptr2=iptr1+8;
printf("\n iptr1=%u, *iptr1=%d, iptr2=%u,
*iptr2=%d", iptr1, *iptr1,iptr2,*iptr2);
}
```

4. Chestiuni de studiat

4.1 Studierea şi însuşirea noțiunilor teoretice şi a exemplelor prezentate.

4.2 Identificarea elementelor de limbaj şi a algoritmilor utilizați.

Lucrarea nr. 4

TABLOURI ŞI POINTERI ÎN LIMBAJUL C

1. Scopul lucrării

Lucrarea are ca scop prezentarea legăturii dintre tablouri şi pointeri în limbajul C

2. Noțiuni teoretice

2.1 Tablouri şi şiruri de caractere

Un tablou este o structură omogenă, formată dintr-un număr finit de elemente de acelaşi tip denumit tipul de bază al tabloului. Sintaxa de bază în declararea unui tablou este:

tip nume_tablou[nr_elem]={val_initiala….};

De exemplu:

int tab[5]={5,4,3,2,1};

defineşte un tablou de 5 valori de tip **int** şi realizează inițializarea acestuia cu valorile din interiorul acoladelor.

Pentru identificarea unui element al unui tablou se foloseşte numele tabloului şi indexul (poziția elementului în tablou). Valorile pe care le poate lua indexul pleacă de la 0, ultimul element având indexul **nr_elem-1**:

Astfel, în exemplul precedent, **tab[0]** este primul element al tabloului şi are valoarea **5**.

Limbajul C nu are un tip de date special pentru şiruri de caractere, dar permite folosirea tablourilor unidimensionale de tip caracter (**char**). Sintaxa de declarare este:

char nume_sir [nr_elem];

Pentru a marca sfârşitul unui şir cu **n** elemente, după ultimul caracter, compilatorul rezervă **n+1** locații de memorie, pe ultima poziție adăugând un octet cu valoarea 0 (caracterul **'\0'**). Astfel, acest terminator **'\0'** permite testarea sfârşitului şirului.

În biblioteca limbajului C există un set de funcții dedicate operațiilor cu şiruri.

Astfel, în fişierul **stdio.h** sunt declarate:

- funcția **gets(sir_dest)** care citeşte caractere introduse de la tastatura şi le transferă în şirul **sir_dest** şi
- funcția **puts(sir)** care afişează şirul **şir** pe ecran ,

iar, în fişierul **string.h,** se găsesc câteva funcții ce realizează operații uzuale cu şiruri, cum ar fi:

- funcția **strcpy(sir_dest,sir_sursa)** care copiază şirul **sir_sursa** în şirul **sir_dest**,
- funcția **strcat(sir1, sir2)** care adaugă şirul **sir2** la sfârşitul şirului **sir1** (concatenare) şi
- funcția **strlen(sir)** care returnează numărul de elemente al şirului **sir**.
- funcția **strcmp(sir1,sir2)** care compară succesiv caracterele celor 2 şiruri. Dacă şirurile sunt identice returnează valoarea 0, iar dacă diferă, o valoare nenulă.

Utilizare acestor funcții este prezentată în exemplul următor, care testează introducerea parolei corecte "*next*":

```
#include <stdio.h>
#include <string.h>
#include<process.h>
void main(void) {
   char sir[20], parola[10];
   strcpy(parola,"next");
   puts("Introduceti parola:");
   gets(sir);
   if (strcmp(sir, parola)) {
      puts("Incorect");
      exit(1);   /*iesire din program */
   }
   else puts("Corect!");
   /*…si se poate executa in continuare programul
*/
}
```

În cazul tablourilor unidimensionale, se poate omite dimensiunea la declarație. În această situație, dimensiunea zonei de memorie alocate este fixată de compilator pe baza listei de constante utilizate la inițializare.

În cazul tablourilor mari, nu mai este necesară numărarea elementelor, ca în declarația:

char sir[]="Nu mai este nevoie de dimensiunea sirului";

2.2 Tablouri şi pointeri

Numele unui tablou fără index este echivalent cu un **pointer constant** de tipul elementelor tabloului, având ca valoare adresa primului element din tablou. Totuşi, în timp ce unei variabile de tip pointer i se atribuie valori la execuție, nu este posibil şi pentru numele unui tablou, care va avea mereu ca valoare adresa primului element. De aceea se spune că numele unui tablou este un pointer constant.

De exemplu, după declarațiile:

```
……..
float ftab[10],*fptr;
int i;
…….
```

se poate face atribuirea:

fptr=ftab;

în urma căreia variabila pointer **fptr** va avea adresa primului element al tabloului **ftab.** Există de asemenea următoarele echivalențe:

1. **&ftab[0] <= => ftab**
2. **&ftab[1] <= => ftab+1**
3. **&ftab[i] <= => ftab+i**
4. **ftab[0] <= => *ftab**
5. **ftab[4] <= => *(ftab+4)**
6. **fptr+i <= => fptr[i]**

Pe baza acestor egalități se poate calcula expresia:

(&ftab[i]-ftab)= = ftab+i- ftab==i

Chiar dacă conține adresa primului element, numele tabloului (fără index) referă întregul tablou, astfel că în exemplul nostru **sizeof(ftab)** va avea valoarea **40** (**10** elemente de **4** octeți fiecare).

În cazul tablourilor multidimensionale, deoarece reprezintă tablouri cu elemente tablouri, numele unui astfel de tablou (fără index) este un pointer de tablouri.

În exemplul:

```
……..
float fmat [10][10];
float *fp;
fp=mat;
……..
```

atribuirea **fp=mat;** este incorectă, deoarece **mat** este pointer de tablouri **float** cu 10 elemente şi nu un pointer **float**.

Astfel, **mat** referă prima linie a matricii identificată prin **mat[0]**, iar **mat[0]** referă primul element al matricii **mat[0][0]**. Pot fi scrise echivalențele:

1. **&mat[0] <= => mat**
2. **&mat[0][0] <= => mat[0]**
3. **mat[0][0] <= => *mat[0] <= => **mat**
4. ***(mat+i) <= => mat[i] <= =>&mat[i][0]**
5. ***(*(mat+1)+5)<==>*(mat[1]+5)<==>mat[1][5]**

În general este valabilă echivalența:
mat[i][j] <= =>*(*(mat+i)+j)

3. Probleme rezolvate

*3.1 Acest program încarcă tabloul **t1** cu numerele 1….10 şi apoi copiază conținutul lui **t1** în tabloul **t2:***

PROGRAMUL 3.1

```
#include <stdio.h>
main ( )
{
  int t1 [10] ,t2[10];
  int i;
  for (i=1;i<11;i++)   t1[i-1] = i;
  for (i=0;i<10;i++)   t2[i] =t1[i];
  for (i=0;i<10;i++)
      printf("%d  ",t2[i] );
}
```

3.2 Acest program calculează urma unei matrice pătrate (suma elementelor de pe diagonala principală) utilizând variabile pointer pentru adresarea elementelor matricei. Aceste variabile pointer sunt inițializat (pentru fiecare linie) cu adresa de început a liniei respective.

PROGRAMUL 3.2

```
 #include <stdio.h>
#include <conio.h>
main ( )
{
int mat[10][10];
int s=0, *ptr,n,i,j;
printf("Dati dimensiunea matricei patrate:");
scanf("%d",&n);
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
   {
     printf("mat[%d][%d]=",i+1,j+1);
     scanf("%d",&mat[i][j]);
   }
for(i=0;i<n;i++)
{
  ptr=mat[i];
  s=s+*(ptr+i);
}
printf("Suma=%d\n",s);
}
```

3.3 Acest program numără spațiile dintr-un şir introdus de la tastatura de către utilizator. Este testat fiecare caracter, iar dacă acesta nu este spațiu, instrucțiunea ***continue*** *forțează reluarea ciclului* ***for****. În cazul în care este găsit un spațiu, valoarea variabilei* ***spațiu*** *este incrementată. Parcurgerea şirului se realizează prin incrementarea variabilei* ***str*** *de tip pointer la un şir de caractere*

PROGRAMUL 3.3

```
#include <stdio.h>
void main (void)
{
   char sir[80], *str;
   int spatiu;
   printf("Introduceti un sir: ");
   gets(sir);
   str = sir;
   for (spatiu=0; *str; str++)
   {
      if (*str != ' ') continue;
      spatiu++;
   }
   printf("Sirul contine %d spatii \n",spatiu);
}
```

4. Chestiuni de studiat

4.1 Studierea noțiunilor teoretice şi a exemplelor prezentate.

4.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizați.

Lucrarea nr. 5

FUNCȚII ÎN LIMBAJUL C

1. Scopul lucrării

Lucrarea are ca scop prezentarea funcțiilor în limbajul **C.**

2. Noțiuni teoretice

În general, un program C este alcătuit din una sau mai multe funcții. Întotdeauna există cel puțin o funcție, funcția **main()** care este apelată la lansarea în execuție a programului.

Sintaxa generală a definirii unei funcții este :

tip_fct nume_fct(listă_declarații_parametrii)
{
<lista_declarații_locale>
listă_instrucțiuni
}

tip_fct este tipul rezultatului returnat de funcție. Dacă o funcție nu întoarce un rezultat, tipul folosit este **void**.

În exemplul următor funcția **max** primeşte doi parametrii de tip **float** şi afişează valoarea maximă şi media acestora. Deoarece funcția nu întoarce niciun rezultat, tipul său este **void**:

Ex.1:

```
…….
void max(float a1,float a2)
{
float maxim;     /* declaratie locală */
if(a1>a2) maxim=a1;
else maxim=a2;
printf
(“Maxim=%f;Medie=%f\n”,maxim,(a1+a2)/2);
a1=7.5;
}
main()
{  …
   float r;
   …
max(r,4.53); /*apel al functiei afmax*/
}
```

Formatul general al apelului unei funcții este:

nume_fct (param1, param2…)

Numim **parametrii formali** identificatorii din **lista_declarații_parametrii** din definiția funcției şi **parametrii efectivi** acele variabile, constante sau expresii din lista unui apel al funcției.

Se observă ca parametrii formali reprezintă variabilele locale ale funcției. Timpul lor de viață corespunde duratei de execuție a funcției.

Transmiterea parametrilor (în urma unui apel al funcției) se realizează prin încărcarea valorii parametrilor efectivi în zona de memorie a parametrilor formali. Acest procedeu se numeşte **transfer prin** valoare.

În cazul în care parametrul efectiv este o variabilă, operațiile efectuate în cadrul funcției asupra parametrului formal nu o afectează. În Ex.1, atribuirea **a1=7.5** din finalul funcției nu modifică valoarea variabilei **r** din apelul **max(r,4.53);** .

Dacă se doreşte modificarea valorii unei variabile indicate ca parametru efectiv într-o funcție, trebuie ca parametrul formal să fie de tip pointer. La apelare, trebuie să i se ofere explicit adresa unei variabile. Acest procedeu se numeşte **transfer prin referință.**

În cazul transferului prin referință, modificarea realizată de funcție asupra parametrilor efectivi este valabilă atât în interiorul cât şi în exteriorul funcției.

Atunci când se doreşte ca funcția să returneze un rezultat se foloseşte instrucțiunea **return** cu sintaxa:

return (expresie)

Valoarea expresiei este rezultatul întors de funcție, iar parantezele sunt opționale.

3. Probleme rezolvate

3.1 Următorul program creează şi implementează o funcţie care caută un caracter într-un şir şi returnează toate poziţiile pe care acesta este găsit. Poziţiile returnate sunt grupate într-un tablou, deoarece rezultatul funcţiei este de tip pointer la întreg.

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
#include<alloc.h> /*<malloc.h> ptr Visual C*/

int *find(char*sir,char caracter)
  {
       int*a,*b;
       char*sir1;
       sir1=sir;
       a=(int*)malloc(strlen(sir));
       b=a;
       while(*sir1!='\0')
           {
              if(*sir1==caracter)
                 {
                    *a=(sir1-sir);
                     a++;
                 }
           sir1++;
       }
    *a=-1;
     return(b);
  }
void main(void)
{
   clrscr();
   char *s="acesta este sirul care va fi analizat";
   char car='a';
   int *pozitia,i;
   pozitia=find(s,car);
   i=0;
   while (pozitia[i] !=-1)
   printf(" %d  ", pozitia[i++]+1);
}
```

3.2 Următorul program calculează suma elementelor unui vector utilizând o funcţie având printre parametrii formali şi o variabilă pointer.

```
#include<stdio.h>
double fsuma(double *ptr, int n)
{
int i;
double sum=0;
for(i=0;i<n;i++)
sum=sum+*(ptr+i);
return sum;
}
main()
{
double tab[20],elem,suma;
int i,nr;
printf("Dati numarul de elemente al vectorului:");
scanf("%d",&nr);
for (i=0; i<nr;i++)
  {
  printf("Numar(%d)=",i+1);
  scanf("%lf", &elem);
  tab[i]=elem;
  }
suma=fsuma(tab,nr);
printf("Suma elementelor vectorului este: %5.2lf",
suma);
 }
```

4. Probleme propuse

4.1. Să se implementeze o funcţie care caută un caracter într-un şir şi returnează de câte ori s-a găsit caracterul.

4.2 Să se implementeze o funcţie care calculează suma elementelor de pe diagonala principală a unei matrice pătrate, utilizând variabile pointer pentru adresarea elementelor matricei.

5. Chestiuni de studiat

5.1 Studierea noţiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizaţi.

5.3 Rezolvarea problemelor propuse

Lucrarea nr.6

STRUCTURI

1. Scopul lucrării

Lucrarea are ca scop prezentarea utilizării tipurilor de date de tip structură în limbajul C.

2. Noțiuni teoretice

Limbajul C permite utilizatorului să definească noi tipuri de date pentru a avea o reprezentare mai convenabilă a informației. Există posibilitatea grupării unor elemente, pentru a reprezenta date complexe, cel mai des sub forma unor structuri.

Structura este o colecție de variabile ce pot fi de tipuri diferite, grupate împreună sub un singur nume şi memorată într-o zonă continuă de memorie.

Sintaxa generală de declarare a unei structuri este:

```
struct      nume_structura  {
   tip1   elem1;
   tip2   elem2;
   …
   tipn   elemn;
     }    lista_var_struct;
```

în care:

struct = cuvânt cheie asociat structurilor;

nume_structura = numele dat de utilizator structurii;

tipi = tipul elementului **elemi** ;

lista_var_struct = lista variabilelor de tip structură **nume_structura** definită anterior.

Elementele componente ale unei structuri se numesc membrii sau câmpuri.

De exemplu un punct identificat prin coordonatele sale **x** şi **y** poate fi reprezentat prin intermediul unei structuri cu două câmpuri **x** şi **y** de tip **float**:

```
struct punct
   {
      float  x;
      float  y;
    }m,n;
```

iar **m** şi **n** sunt două variabile de tip structură punct.

Referirea la un membru al unei structuri se face cu ajutorul operatorului punct „ **.** ” plasat între numele structurii şi numele membrului, **m.x** fiind abscisa punctului **m**.

Inițializarea unei variabile de tip structură se poate face:

- prin inițializarea fiecărui membru al structurii sau,
- global, prin enumerarea, între acolade, a valorilor inițiale ale membrilor, în ordinea în care apar în declarație.

Pentru variabilele structură **punct** din exemplul precedent se pot face inițializările:

```
m.x=10.5;
m.y=m.x+7;
n={12.3, 34.5};
```

Pot fi definite şi structuri încuibate. De exemplu, un dreptunghi poate fi definit prin două puncte ale unei diagonale:

```
struct dreptunghi {
   struct punct pt1;
   struct punct pt2;
      };
struct dreptunghi d;
d.pt2.y=9.7;
```

În ultima atribuire, ordonata punctului **pt2** al dreptunghiului **d** primeşte valoarea **9.7.**

De asemenea, pot fi declarate tablouri cu elemente structuri:

```
struct punct sirpuncte[10];
```

Pentru acest şir de puncte, secvența:

```
sirpuncte[2].x=20;
```

atribuie abscisei celui de-al treilea punct valoarea **20**.

Pot fi efectuate operații de atribuire între variabile de tip structură de acelaşi tip, care echivalează cu atribuiri membru cu membru.

Pot fi definite şi variabile pointer de tip structură:

```
struct punct *pct;
pct=&sirpuncte[5];
```

Pentru accesul la un element al unei structuri indicate de un pointer se utilizează **operatorul de selecție indirectă:'->'** (săgeata):

```
pct->y=12.3;
```

3. Probleme rezolvate

3.1 *Se consideră o grupă de maxim 20 de studenți identificați prin numele lor. Pentru fiecare student se cunosc notele la cele patru examene din sesiune. Să se afişeze toți studenții bursieri (a căror medie este mai mare sau egală cu 8.5).*

```
#include<stdio.h>
struct student
    {
    char nume[15];
    int nota1;
    int nota2;
    int nota3;
    int nota4;
    } stud[20];
int i,n;
float medie;
void main (void)
{
  printf("     Dati numarul de studenti: ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
    {
       printf("    NUME=");
       scanf("%s",stud[i].nume);
       printf("    NOTA1=");
       scanf("%d",&stud[i].nota1);
       printf("    NOTA2=");
       scanf("%d",&stud[i].nota2);
       printf("     NOTA3=");
       scanf("%d",&stud[i].nota3);
       printf("     NOTA4=");
       scanf("%d",&stud[i].nota4);
    }
i=0;
while(i<n)
  {
   medie=(float)(stud[i].nota1+stud[i].nota2+
                 stud[i].nota3+ stud[i].nota4)/4;
   if(medie>=8.5)
       {
       puts(stud[i].nume);
       printf("%5.2f\n",medie);
       }
       i++;
 }
}
```

3.2. *Programul următor utilizează tipul structură asociat cărților dintr-o bibliotecă şi face o selecție după anul apariției.*

```
#include<stdio.h>
#include<conio.h>
struct carte
    {
    char titlu[20];
    char autor[20];
    int an;
    float pret;
    };
void main(void)
{
struct carte bib[50];
int i,n=0;
clrscr();
printf("\n Dati numarul de carti:");
scanf("%d",&n);

for(i=0;i<n;i++)
    {
       printf("    Titlu=");
       scanf("%s",bib[i].titlu);
       printf("    Autor=");
       scanf("%s",bib[i].autor);
       printf("    Anul aparitiei=");
       scanf("%d",&bib[i].an);
       printf("     Pret=");
       scanf("%f",&bib[i].pret);
     }

printf("Carti editate înainte de revolutie:\n");
i=0;

while(i<n)
  {
   if(bib[i].an<=1989)
    {

     puts(bib[i].titlu);
     puts(bib[i].autor);
       printf("\n");
     }
       i++;
    }
}
```

4. Probleme propuse

4.1 Pentru o grupă de studenți se cunosc numele studenților şi 5 note obținute la sfârşitul semestrului. Se cere să se afişeze studenții care nu au nici o restanță.

4.2. Să se folosească structuri de tip punct, pentru a determina daca trei puncte A,B,C (date prin coordonatele lor) pot forma un triunghi şi de ce tip (oarecare, isoscel, echilateral).

4.3 Folosind tipul complex ca o structură cu două câmpuri (parte reală şi parte imaginară), să se simuleze operațiile asupra numerelor complexe: adunare, scădere, înmulțire, împărțire, modul, parte reală şi parte imaginară.

5. Chestiuni de studiat

5.1 Studierea noțiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizați.

5.3 Rezolvarea problemelor propuse.

Lucrarea nr. 7

LISTE

1. Scopul lucrării

Lucrarea are ca scop prezentarea listelor în limbajul C, punând accentul pe reprezentarea acestora cu ajutorul structurilor de date dinamice.

2. Noțiuni teoretice

În anumite situații este necesară organizarea informațiilor sub forma unor liste. De exemplu lista persoanelor dintr-o instituție, a produselor dintr-un magazin etc.. Lista este deci compusă din elemente de acelaşi tip, iar informația conținută în fiecare element al listei este de multe ori complexă.

Lista are un număr variabil de elemente şi deci un caracter dinamic. Astfel, la începutul execuției programului, lista este goală urmând ca aceasta să fie completată şi să i se aducă apoi modificări, tot prin program.

Limbajul C permite implementarea listelor cu ajutorul variabilelor dinamice de tip structură. Utilizarea tablourilor pentru reprezentarea unor liste este posibilă, dar nu este eficientă, deoarece listele au un caracter dinamic spre deosebire de tablouri. De aceea, practica uzuală este de a ordona elementele unei liste folosind variabile pointer care intră în componența elementelor. Utilizarea acestor variabile pointer dă un caracter recursiv elementelor listei. Listele implementate astfel se numesc *înlănțuite*.

Elementele unei astfel de liste poartă denumirea uzuală de *noduri*. Dacă între noduri există o singură relație de legătură lista se numeşte *simplu înlănțuită***,** iar dacă există două relații de legătură lista se numeşte *dublu înlănțuită*.

Cele mai uzuale operații care se pot efectua asupra unei liste înlănțuite sunt: crearea listei, accesul la un nod, adăugarea, ştergerea sau modificarea unui element şi ştergerea listei.

2.1. Liste simplu înlănțuite

Între nodurile unei liste simplu înlănțuite există o singură relație de legătură, de obicei de indicare a succesorului unui element. Această legătură se implementează cu ajutorul unui pointer ce memorează adresa nodului următor din listă.

De exemplu pentru o listă de persoane simplu înlănțuită, la care se cunosc numele, prenumele şi vârsta, nodul are următoarea formă:

```
struct nod {
        char  nume[15];
        char  prenume[15];
        int varsta;
        struct  nod *urm;};
```

În această structură, câmpul **urm** va conține adresa următorului nod din listă. El este un pointer la o variabilă de tip structură identică cu cea din care face parte. Pentru ultimul nod din listă, variabila **urm** va avea valoarea **NULL**, deoarece nu mai urmează un alt nod. De asemenea, spre primul nod al listei nu pointează nici un alt nod.

Cunoaşterea primului şi ultimului element al listei este importanță, deoarece permite accesul la orice element prin parcurgerea listei (începând cu primul) şi facilitează operațiile de adăugare de elemente noi la sfârşitul listei.

Atunci când trebuie adăugat un element în listă se parcurg etapele:

1. se alocă dinamic spațiu pentru respectivul element,
2. se creează elementul prin înscrierea informațiilor corespunzătoare şi
3. se leagă în listă.

Când un element trebuie scos din listă, se rup şi se refac legăturile, iar spațiul ocupat de acesta se eliberează.

2.2.Liste dublu înlănțuite

Între nodurile unei liste dublu înlănțuite vor exista două relații de legătură, cu elementul anterior şi cu elementul următor. În acest caz, vor exista doi pointeri de legătură ce memorează adresa nodului anterior şi respectiv a celui următor din listă. Implementarea exemplului precedent se va face, în cazul utilizării listelor dublu înlănțuite, sub forma:

```
struct nod
        {
        char  nume[15];
        char  prenume[15];
        int varsta;
        struct  nod *ant;
        struct  nod *urm;
        };
```

Adăugarea unui element într-o listă dublu înlănțuită va necesita aceleaşi etape ca în cazul listelor simplu înlănțuite. Funcția care realizează adăugarea unui element într-o listă dublu înlănțuită, al cărui nod are structura de mai sus, va avea următoarea formă:

```
void adauga(char *Nume,char *Prenume,int Varsta)
 {
  struct nod *p;
  p=(struct nod *)malloc(sizeof(struct nod));
  if(p==NULL)
     {
     printf("Memorie insuficientă.\n");
     return;
      }
  strcpy(p->nume,Nume);
  strcpy(p->prenume,Prenume);
  p->varsta=Varsta;
  p->urm=NULL;
  ultim->urm=p;
  p->ant=ultim;
  ultim=p;
 }
```

3. Problemă rezolvată

Să se creeze o listă simplu înlănțuită a persoanelor dintr-o instituție în care să se memoreze numele, prenumele şi vârsta fiecăruia. Să se afişeze persoanele a căror vârstă este mai mică de 40 de ani. Persoanele care au vârsta de pensionare (65 de ani) vor fi eliminate din listă. Se va afişa apoi lista persoanelor rămase.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <conio.h>
struct nod {
            char nume[15];
            char prenume[15];
            int varsta;
            struct nod *urm;};
struct nod *prim,*ultim;

void adauga(char *Nume,char *Prenume,int Varsta)
 {
  struct nod *p;
  p=(struct nod *)malloc(sizeof(struct nod));
  if(p==NULL)
     {
     printf("Memorie insuficientă.\n");
     return;
      }
  strcpy(p->nume,Nume); strcpy(p->prenume,Prenume);
  p->varsta=Varsta;
  p->urm=NULL;
  if(prim==NULL) prim=ultim=p;
      else
      {
            ultim->urm=p;
            ultim=p;
      }
 }
void sterge(struct nod *p)
  {
   struct  nod *q;
   if(!p) return;
   if(prim==p)
    {
     prim=p->urm;
     if(prim==NULL) ultim=NULL;
     }
        else {
 for(q=prim;q->urm!=p&&q->urm!=NULL;q=q->urm)
        if(q->urm==NULL)
                {
                printf("Elementul nu aparține listei.\n");
                return;
                }
           q->urm=p->urm;
           if(p==ultim) ultim=q;
               }
  free(p);
 }
void main(void) {
      char Nume[15],Prenume[15];
      int Varsta;
      int i,n;
      struct nod *p,*q;
      printf("Numărul de persoane:");scanf("%d",&n);
      prim=ultim=NULL;
      for(i=0;i<n;i++)
      {
            printf("Nume:");gets(Nume);
            printf("Prenume:");gets(Prenume);
            printf("Varsta:");scanf("%d",&Varsta);
            adauga(Nume,Prenume,Varsta);
       }
       printf("Lista persoanelor sub 40 de ani :\n");
       for(p=prim;p!=NULL;p=p->urm)
         if(p->varsta <= 40) printf("%15s%15s - %d\n",
            p->nume,p->prenume,p->varsta);
            p=prim;
       while(p!=NULL) {
            if(p->varsta > 65)
              {
                 q=p->urm;
                 sterge(p);
                 p=q;
               }
            else p=p->urm;
       }
       printf("Lista persoanelor ramase:\n");
       for(p=prim;p!=NULL;p=p->urm)
    printf("%15s%15s - %d\n",p->nume,p-> prenume,
p->varsta);
}
```

4. Probleme propuse

4.1 Să se implementeze lista din problema rezolvată 3 cu ajutorul listelor dublu înlănțuite.

4.2 Să se creeze o listă a studenților prezenți la un examen de admitere în care să se memoreze numele, prenumele şi media de admitere. Să se afişeze studenții care primesc bursă (cu media peste 9.50). Studenții cu media de admitere sub 5.00 sunt considerați respinşi şi vor fi eliminați din listă. Se va afişa apoi lista studenților admişi.

5. Chestiuni de studiat

5.1 Studierea noțiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizați.

5.3 Rezolvarea problemelor propuse

Lucrarea nr.8

FIŞIERE IN LIMBAJUL C

1. Scopul lucrării

Lucrarea are ca scop prezentarea utilizării fişierelor în limbajul C.

2. Noţiuni teoretice

Un fişier reprezintă o colecţie ordonată de înregistrări. Majoritatea operaţiilor de intrare–ieşire se bazează pe manipularea fişierelor. Datele introduse de la tastatură formează un fişier de intrare, în timp ce datele afişate pe display sau listate la imprimantă formează un fişier de ieşire.

Există două tipuri de fişiere: text şi binare. În timp ce fişierele text conţin caractere ASCII în gama 0-127 (informaţie citibilă), fişierele binare conţin înşiruiri de caractere, neinteligibile pentru utilizator. De exemplu, fişierele sursă sunt fişiere text, în timp ce fişierele executabile sunt fişiere binare.

Prelucrarea unui fişier presupune o serie de operaţii precedate de deschiderea fişierului şi finalizate cu închiderea acestuia.

În urma deschiderii unui fişier se generează un pointer la o structură de tip **FILE** predeclarată în **stdio.h**. Sintaxa de declarare a unui pointer la **FILE** este: **FILE*fptr;**

fptr fiind numele variabilei pointer cu care se lucrează în continuare.

Deschiderea unui fişier se realizează cu funcţia **fopen()** cu sintaxa generală:

```
FILE *fopen(const char*nume_fisier ,
            const char *mod);
```

în care:

nume_fisier este numele fişierului care se va deschide sau crea;

mod este modul în care este deschis fişierul:

"r" deschide fişierul pentru citire;
"w" deschide un fişier pentru scriere;
"a" deschide sau creează un fişier pentru scriere la sfârşitul fişierului (adăugare);
"r+" deschide un fişier pentru actualizare (citire + scriere);
"w+" deschide un fişier pentru actualizare, conţinutul anterior se elimină;
"a+" deschide un fişier pentru actualizare la sfârşit.

Fişierele pot fi deschise în mod binar sau text, după cum este specificat în argumentul **mod** al funcţiei **fopen** prin adăugarea literei **t** pentru text sau **b** pentru binar.

Dacă operaţia de deschidere are succes, funcţia returnează un pointer la **FILE** (pointer care va fi folosit în continuare în operaţiile asupra fişierului), iar dacă eşuează, **fopen** întoarce valoarea **NULL**.

În exemplul următor:

```
FILE *fptr1, *fptr2;
fptr1=fopen("fist.txt","r+t");
fptr2=fopen("fisb.bin","wb");
```

sunt deschise două fişiere: primul de tip text pentru operaţii de actualizare şi cel de-al doilea de tip binar pentru scriere.

Închiderea unui fişier se realizează cu funcţia **fclose()** având sintaxa generală:

```
int fclose(FILE* fptr_fisier);
```

care va închide fişierul specificat de **fptr_fisier**.

Funcţia **fclose()** returnează valoarea 0 în caz de închidere cu succes a fişierului şi EOF dacă a apărut o eroare.

Închiderea fişierelor din exemplul precedent se va face cu secvenţa:

```
fclose(fptr1);
fclose(fptr2);
```

Scrierea de date în fişier se realizează cu ajutorul funcţiei **fprintf()**:

```
int fprintf(FILE *fptr, const char *format,
            arg1, arg2,…,argn)
```

care scrie în fişierul pointat de **fptr** datele specificate de **arg1…argn**, în formatul specificat prin şirul de caractere **format.**

În exemplul:

```
FILE *fpt;
int i=5;
char c='B';
float f=2.7543;
fpt=fopen("fis.dat","w");
fprintf(fpt,"%d,%c,%f",i,c,f);
```

prin utilizarea funcţiei **fprintf(),** se scriu în fişierul **fis.dat**, descris de **fpt,** un întreg, un caracter şi o variabilă de tip **float**.

Citirea de date dintr-un fişier se realizează cu ajutorul funcţiei **fscanf():**

```
int fscanf(FILE *fptr, const char *format,
           arg1, arg2,…,argn)
```

care citeşte din fişierul indicat de **fptr** date sub controlul formatului specificat în **format** şi le atribuie variabilelor prin adresele specificate în **arg1, arg2,……argn,** care, de această dată, sunt pointeri.

3. Problemă rezolvată

Să se creeze un fişier text ce conţine informaţii despre produsele dintr-un magazin. Să se scrie apoi o funcţie de adăugare, apoi una de căutare în fişier după nume şi modificarea numărului de bucăţi . În final să se listeze conţinutul fişierului modificat.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<process.h>
#define nf "produse.txt"
typedef struct {
    char nume[10];
    int nr;
    float pret;
     }prod;
FILE *fp;
void creare ()
{
   if((fp=fopen(nf,"w"))==NULL)
        {
            printf("Eroare de creare\n");
            exit(1);
        }
        fclose(fp);
}
void listare ()
{
    prod s;
    clrscr();
    if((fp=fopen(nf,"r"))==NULL)
            {
                printf("Eroare de deschidere \n");
                exit(1);
            }
        do
            {
                fread(&s,sizeof(prod),1,fp);
                if(feof(fp)) break;
                printf("\n%s",s.nume);
                printf("\n%d",s.nr);
                printf("\n%5.2f",s.pret);
            }
        while (!feof(fp));
        fclose(fp);
}
void adaugare ()
{
    char c;
    prod s;
    clrscr();
    if((fp=fopen(nf,"a"))==NULL)
        {
        printf("Eroare de deschidere\n");
        exit(1);
        }
        c='d';
    while(c=='d')
    {
        printf("\n Nume:");
        scanf("%s",s.nume);
        printf("\nNumarul de bucati:");
        scanf("%d",&(s.nr));
        printf("\nPret: ");
        scanf("%f",&(s.pret));
        fwrite(&s,sizeof(prod),1,fp);
         printf("\n\n  Mai doriţi adăugare?  (d/n):  ");
        c=getch();
     }
        fclose(fp);
 }
void modificare()
{   prod s;
    long int poz;
    int gasit=0;
    char n[10];
    int nrnou;
    printf("\n  Dati numele dupa care se cauta: ");
    scanf("%s",n);
    printf("\n Dati noua cantitate: ");
        scanf("%d",&nrnou);
        if((fp=fopen(nf,"r+"))==NULL)
            {
                printf("Eroare de deschidere\n");
                exit(1);
            }
        while((!gasit)&&(!feof(fp)))
        {
            poz=ftell(fp);
            fread(&s,sizeof(prod),1,fp);
           if(!strcmp(n,s.nume))
             gasit++;
        }
        if(!gasit)
            {
            printf("\nNu s-a gasit inregistrarea ");
            getch();
            }
        else
            {
            fseek(fp,poz,SEEK_SET);

            fread(&s,sizeof(prod),1,fp);
            s.nr=nrnou;
             fseek(fp,poz,SEEK_SET);
            fwrite(&s,sizeof(prod),1,fp);
        }
            fclose(fp);
    }
void main()
    {  creare();
        adaugare();
        listare();
        modificare();
        listare();  }
```

4. Probleme propuse

4. 1. Să se creeze şi să se listeze un fişier binar cu studenţi, în care să se memoreze numele şi două note.

Datele se citesc dintr-un fişier text creat anterior.

4.2. Să se scrie o procedură de creare a unui fişier text ce conţine nume de studenţi şi o alta de actualizare prin adăugare a acestui fişier. În final se va scrie o procedură de listare a conţinutului fişierului.

5. Chestiuni de studiat

5.1 Studierea noţiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemei rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizaţi.

5.3 Rezolvarea problemelor propuse

Lucrarea nr. 9

REZOLVAREA ECUAȚIILOR

1. Scopul lucrării

Lucrarea are ca scop prezentarea unei metode numerice de rezolvare a ecuațiilor algebrice şi transcendente cu ajutorul limbajului C++.

2. Noțiuni teoretice

Există mai multe metode numerice utilizate pentru calculul rădăcinilor reale ale unei ecuații algebrice: metoda bisecției, metoda poziției false, metoda aproximațiilor succesive, metoda lui Newton, metoda lui Bairstow, etc..

Acestea sunt metode de calcul care presupun utilizarea unor algoritmi numerici ce permit găsirea rădăcinilor. Înainte de calculul propriu-zis al rădăcinilor (prin procedee iterative), trebuie realizată o separarea a rădăcinilor ce constă în găsirea acelor intervale care conțin cel mult o rădăcină.

Metoda bisecției

Această metodă mai este numită metoda înjumătățirii intervalului sau metoda bipartiției.

Metoda permite găsirea unei rădăcini (cu o anumită eroare ε) a ecuației:

$$f(x)=0$$

în intervalul **[a,b]**, unde **f : [a,b] —>R** şi **f** este continuă pe **[a,b]**. Se presupune că s-a realizat în prealabil o separare a rădăcinilor, astfel încât pe intervalul **[a,b]** există cel mult o rădăcină ξ.

Algoritmul începe prin analizarea următoarelor patru situații:

1. $f(a)=0$ şi deci $\xi=a$;
2. $f(b)=0$ şi deci $\xi=b$;
3. $f(a)\cdot f(b)<0$ şi atunci ξ aparține intervalului (a,b)
4. $f(a)\cdot f(b)>0$ şi atunci nu există o rădăcină în intervalului [a,b].

Variantele 1,2 şi 4 presupun încheierea procesului de găsire a rădăcinii.

Algoritmul continuă, în varianta 3, cu înjumătățirea intervalului [a,b] şi determinarea valorii $x_0=(a+b)/2$. Se verifică dacă x_0 este soluție a ecuației, prin evaluarea $|f(x_0)| < \varepsilon$. În caz contrar, se alege semiintervalul $[a_1,b_1]$ la capetele căruia funcția are semne opuse $(f(a_1)\cdot f(b_1)\leq 0)$ şi se repetă paşii de mai sus. Se obțin intervale de tip $[a_i, b_i]$ ca fiind jumătate din intervalul $[a_{i-1}, b_{i-1}]$ prin metoda generală:

$x_{i-1}=(a_{i-1}+b_{i-1})/2;$

$a_i=a_{i-1}, b_i=x_{i-1}$ **dacă** $f(a_{i-1})\cdot f(x_{i-1})<0$

$a_i=x_{i-1}, b_i=b_{i-1}$ **dacă** $f(a_{i-1})\cdot f(x_{i-1})>0$

Se obțin astfel două şiruri convergente:

- şirul a_n al extremităților stângi ale intervalelor, care este monoton crescător ($a < a_1 < \ldots < a_n$)
- şirul b_n al extremităților drepte ale intervalelor, care este monoton descrescător ($b > b_1 > \ldots > b_n$)

Se observă şi că $b_n-a_n=(b-a)/2^n$.

Aşadar, a_n şi b_n vor converge ambele către soluția ξ, deoarece există o valoare **n** pentru care $|b_n-a_n| < \varepsilon$, unde ε este eroarea impusă pentru calculul soluției ecuației date. Se poate aproxima soluția ecuației cu valoarea mijlocului intervalului $[a_n , b_n]$.

În continuare, pe baza algoritmului prezentat, se vor prezenta două funcții în limbajul C++, corespunzătoare rezolvării ecuațiilor algebrice şi respectiv transcendente.

Exemplul 1

```
int bisectiepol (double s, double d, int grad,
                 double coef[], double err, double *rad)
{
double xm ;
if(poly(s,grad,coef)*poly(d,grad,coef)>0) return 0;
if( poly (s,grad,coef) == 0 )
   {
    *rad=s;
    return 1;
    }
if(poly(d,grad,coef)==0)
   {
   *rad=d;
    return 1;
    }
xm=0.5*(s+d);
```

```
while((fabs(d-s)>err) &&(poly(xm,grad,coef)!=0))
    {
    xm=0.5*(d+s);
    if(poly(s,grad,coef)*poly(xm,grad,coef)<0)
        d=xm;
     else s=xm;
     }
*rad=xm;
return 1;
}
```

În acest exemplu **s** şi **d** reprezintă limitele stânga şi respectiv dreapta ale intervalelor de lucru, iar **xm** mijlocul acestora. Este utilizată funcția matematică **poly()** din **math.h,** care permite aflarea valorii unui polinom într-un punct (primul parametru al funcției) cunoscându-se gradul polinomului (al doilea parametru) şi vectorul coeficienților acestuia (al treilea parametru).

Funcția întoarce valoarea 0 în cazul în care nu este găsită o rădăcină în intervalul specificat şi valoarea 1 în caz de succes, valoarea soluției fiind transferată în variabila ***rad**.

Exemplul 2.

```
int bisectiefct(double(*f)(double),double s,
double d, double err, double *rad)
{
double xm;
if(f(s)*f(d)>0) return 0;
if(f(s)==0)
{ *rad=s;
return 1;
}
if(f(d)==0)
  {
  *rad=d;
  return 1;
  }
xm=0.5*(s+d);
while( (fabs(d-s)>err)&&(f(xm)!=0) )
  {
   xm=0.5*(s+d);
   if( f(s)*f(xm)<0) d=xm;
   else s=xm;
   }
*rad=xm;
return 1;
}
```

Implementarea acestei funcții s-a realizat în mod asemănător cu funcția din exemplul 1. Primul parametru al acestei funcții este un pointer ce conține adresa funcției matematice pentru care se caută soluțiile.

3. Probleme rezolvate

3.1 Să se afle dacă ecuația: $x^3-9x^2+23x-15=0$ are o rădăcină în intervalul (-4.5,6), şi să se afle valoarea acesteia (în cazul în care există) cu o eroare de 0.000001.

```
#include<math.h>
#include<iostream.h>
int bisectiepol (double s, double d, int grad,
                 double coef[], double err, double *rad)
{
double xm ;
if(poly(s,grad,coef)*poly(d,grad,coef)>0) return 0;
if( poly (s,grad,coef) == 0 )
   {
    *rad=s;
    return 1;
    }
if(poly(d,grad,coef)==0)
   {
   *rad=d;
    return 1;
    }
xm=0.5*(s+d);
while((fabs(d-s)>err) &&(poly(xm,grad,coef)!=0))
    {
    xm=0.5*(d+s);
    if(poly(s,grad,coef)*poly(xm,grad,coef)<0)
        d=xm;
     else s=xm;
     }
*rad=xm;
return 1;
}

void main(void)
{
double *rad;
double f[]={-15,23,-9,1};
if (bisectiepol(4.5,6,3,f,0.000001,rad)==1)
{cout<<"Functia are o radacina in intervalul (4.5,6)
egala cu:";
cout<<*rad;}
else
cout<<"Functia nu are radacina in intervalul
specificat";
}
```

3.2. Să să afle soluția ecuației transcendente $x - e^{-x} = 0$ *în intervalul (0.1,1), aplicând metoda bisecției cu o eroare de calcul de* $\varepsilon = 0.0000000001$

```
#include<math.h>
#include<iostream.h>
double fct(double x)
 {
 double val_fct;
 val_fct=x-exp(-x);
 return (val_fct);
 }
int bisectiefct(double(*f)(double),double s,
double d, double err, double *rad)
{
double xm;
if(f(s)*f(d)>0) return 0;
if(f(s)==0)
{ *rad=s;
return 1;
}
if(f(d)==0)
  {
  *rad=d;
  return 1;
  }
xm=0.5*(s+d);
while( (fabs(d-s)>err)&&(f(xm)!=0) )
  {
   xm=0.5*(s+d);
   if( f(s)*f(xm)<0) d=xm;
   else s=xm;
   }
*rad=xm;
return 1;
}

void main(void)
{
double s=0.1,d=1.0,err=0.00000001,*sol;
bisectiefct(fct,s,d,err,sol);
cout<<"Solutia ecuatiei f(x)=0 pe intervalul:
("<<s<<","<<d<<") este: "<<*sol;
}
```

4. Probleme propuse

4.1 Să se afle dacă ecuația $x^3 - 6x^2 + 8x = 0$ *are o rădăcină în intervalul (1,3), iar în caz afirmativ să se găsească această soluție.*

4.2 Să se găsească o rădăcină a ecuației $x - \sin(x) - 0.25 = 0$ *în intervalul (1,2) cu o precizie de 0.000001 .*

5. Chestiuni de studiat

5.1 Studierea noțiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizați.

5.3 Rezolvarea problemelor propuse

Lucrarea nr. 10

INTERPOLAREA FUNCȚIILOR

1. Scopul lucrării

Lucrarea are ca scop prezentarea unei metode numerice de aproximare a funcţiilor tabelate şi a implementării acesteia în limbajul C++.

2. Noţiuni teoretice

Interpolarea reprezintă o metodă numerică de aproximare a funcţiilor date sub formă tabelară. Având un set discret de date $[x_i,y_i]$ (i=0,1,…,n) (obţinut în urma unor experimente, măsurători etc.), metoda presupune găsirea unei funcţii **f(x)** continuă care să verifice $y_i=f(x_i)$ (i=0,1,…,n). Funcţia **f(x)** se numeşte funcţie de interpolare, iar punctele x_i noduri ale reţelei de interpolare.

Uzual, funcţia de interpolare are o formă simplă pentru a permite cu uşurinţă aflarea valorilor în orice punct al domeniului de definiţie şi pentru a putea fi uşor prelucrată (derivată, integrată etc.). De aceea, interpolarea este utilizată şi în cadrul metodelor numerice de derivare, integrare etc..

Calculul funcţiei **f(x)** pentru valori ale lui **x** cuprinse între nodurile reţelei de interpolare se numeşte interpolare, iar dacă x se află în afara reţelei, extrapolare.

2.1 Interpolarea polinomială

Cea mai utilizată funcţie de interpolare este funcţia polinomială. Interpolarea polinomială presupune găsirea unui polinom **P(x)** care să verifice:

$$P(x_i)=y_i,\ i=0,1,\ldots,n \tag{1}$$

Dacă polinomul P(x) are expresia:

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \tag{2}$$

condiţiile din relaţia (1) sunt echivalente cu:

$$\begin{cases} a_n x_0^{\ n} + a_{n-1}x_0^{\ n-1} + \ldots + a_1 x_0 + a_0 = y_0 \\ a_n x_1^{\ n} + a_{n-1}x_1^{\ n-1} + \ldots + a_1 x_1 + a_0 = y_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_n x_n^{\ n} + a_{n-1}x_n^{\ n-1} + \ldots + a_1 x_n + a_0 = y_n \end{cases} \tag{3}$$

Deoarece determinantul acestui sistem (de tip Vandermonde):

$$D = \prod_{\substack{i,j=0 \\ j>i}}^{n-1} (x_j - x_i) \tag{4}$$

este diferit de zero (nodurile x_i sunt distincte), sistemul va avea o soluţie unică pentru coeficienţii $a_0, a_1 \ldots .a_n$ şi deci pentru polinomul de interpolare.

Se obţine aşa numitul polinomul de interpolare al lui Lagrange, care are forma:

$$P(x) = \sum_{i=0}^{n} y_i \prod_{j=0,\, j\neq i}^{n} \frac{x - x_i}{x_i - x_j} \tag{5}$$

Deci, cu ajutorul acestui polinom, se poate calcula valoarea funcţiei în orice punct necunoscut cuprins între x_0 si x_n.

Implementarea polinomului de interpolare Lagrange se poate face cu funcţia C++:

```
double lagrange(int n, float x[],
                float y[],float point)
{
 int i,j;
 float sum=0, prod;
 for(i=0; i<n; i++)
 {
  prod = 1;
  for(j=0;j<n;j++)
     if (j!=i)
     prod*=(point-x[j])/(x[i]-x[j]);
  sum+=y[i]*prod;
 }
return sum;
}
```

În cazul în care reţeaua de interpolare are doar două puncte, interpolarea devine liniară:

$$f(x) = y = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1} \tag{6}$$

Ultima egalitate din relaţia (6) reprezintă chiar ecuaţia unei drepte care trece prin punctele (x_1,y_1) şi (x_2,y_2).

3. Problemă rezolvată

În urma unor măsurători s-au obţinut următoarele date memorate în variabilele x şi y:

x	*0*	*0.1*	*0.2*	*0.3*	*0.4*	*0.5*	*0.6*	*0.7*	*0.8*	*0.9*
y	*7*	*14*	*17*	*20*	*22*	*25*	*26*	*28*	*29*	*30*

Se cere să se determine puncte între nodurile reţelei de interpolare (de exemplu pentru: x=0.35 ; x=0.64 ; x=0.89)

```
#include<iostream.h>
double lagrange(int n, float x[],
                float y[],float point)
{
 int i,j;
 float sum=0, prod;
 for(i=0; i<n; i++)
 {
  prod = 1;
  for(j=0;j<n;j++)
      if (j!=i)
      prod*=(point-x[j])/(x[i]-x[j]);
   sum+=y[i]*prod;
  }
return sum;
}

void main(void)
{
int n=10;
float val;
float x[]={0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};
float y[]={7,14,17,20,22,25,26,28,29,30};
float p=0.64;
val=lagrange(n,x,y,p);
cout<<"Valoarea functiei de interpolare in
punctul x="<<p<<" este: "<<val;
}
```

4. Problemă propusă

Plecând de la datele problemei rezolvate 3, să se scrie un program care realizează interpolarea, însă pentru un număr mai mic de noduri ale reţelei de interpolare. De exemplu, pentru 7 noduri alese în mod diferit: uniform distribuit, mai multe în prima parte sau mai multe în a doua parte. Să se compare rezultatele obţinute pentru aflarea valorilor funcţiei în punctele x=0.35, x=0.64 şi x=0.89 . Să se comenteze rezultatele obţinute.

5. Chestiuni de studiat

5.1 Studierea noţiunilor teoretice şi a exemplelor prezentate.

5.2 Studierea problemelor rezolvate şi identificarea elementelor de limbaj şi a algoritmilor utilizaţi.

5.3 Rezolvarea problemei propuse.