LIMBAJE DE PROGRAMARE

CUPRINS

LIMBAJE DE PROGRAMARE

INTRODUCERE

Disciplina *Limbaje de programare* este una din disciplinele fundamentale din planul de învățământ de la specializarea Electromecanică - Frecvență Redusă şi are rolul de a prezenta studenților elemente de programare în limbajul C++. Studiul acestor noțiuni este condiționat de parcurgerea elementelor prezentate la disciplina *Programarea calculatoarelor,* care se studiază în semestrul precedent.

Acest manual universitar este structurat pe 8 unități de învățare:

1. Elemente generale ale limbajului C
2. Variabile pointer
3. Tablouri
4. Structuri
5. Liste
6. Fişiere
7. Completări aduse de C++
8. Rezolvarea problemelor de analiză numerică în limbajul C++

Fiecare unitatea de învățare poate fi parcursă în 3-4 ore de studiu individual.

În afară de studiul individual bazat pe aceste unități de învățare, disciplina *Limbaje de programare* are prevăzute şi două ore de activitate de laborator pe săptămână. Lucrările de laborator reprezintă aplicații practice ale noțiunilor prezentate în unitățile de învățare şi se desfăşoară la rețelele de calculatoare ale Facultății de Inginerie Electrică. Exemplele din cadrul unităților de învățare pot fi editate, compilate şi executate la aceste rețele de calculatoare sau chiar pe calculatoarele personale, necesitând variante simple, gratuite, de compilatoare pentru limbajul C, unul dintre ele fiind prezentat şi în prima unitate.

Competențele generale dobândite de studenți în urma parcurgerii unităților de învățare şi a efectuării şedințelor de laborator sunt legate de utilizarea variabilelor pointer, a tipurilor de date definite de utilizator, a listelor, a fişierelor şi de implementarea unor metode de calcul numeric în limbajul C.

Astfel, legat de variabilele pointer, studentul va cunoaşte sintaxa de declarare, regulile specifice operațiilor cu pointeri, conexiunea cu tablourile şi gestionarea memoriei alocate dinamic.

În ceea ce priveşte tipurile definite de utilizator, parcurgerea manualului oferă informații ce permit cunoaşterea şi manipularea structurilor şi a listelor simplu şi dublu înlănțuite.

Relativ la elementele legate de utilizarea limbajului C, prezentate în unitățile anterioare, în unitatea de învățare 7 sunt prezentate completările aduse de limbajul C++, menite să ofere competențe în elaborarea de programe sursă în ambele variante.

În ultima unitate de învățare sunt prezentate principiile şi implementările în limbajul C++ a cinci metode numerice, care oferă exemple suplimentare de utilizare a elementelor de limbaj şi familiarizează studentul cu aplicații utile în calculele cu caracter ingineresc.

Pentru fiecare unitate de învățare vor fi parcurse noțiunile teoretice, vor fi studiate şi înțelese exemplele prezentate. O bună înțelegere a programelor sursă presupune parcurgerea acestora linie cu linie şi eventual compilarea şi execuția acestora. Unitățile conțin teste de evaluare însoțite de răspunsuri, precum şi lucrări de verificare a căror rezolvare constă în elaborarea unor programe sursă asemănătoare celor prezentate în manual.

Bibliografia unităților de învățare conține câteva poziții ce se regăsesc în biblioteca facultăților cu profil electric, dar orice manual sau resursă web legată de utilizarea limbajului C++ poate fi utilă pentru aprofundarea noțiunilor prezentate.

Activitatea de laborator se finalizează cu un test ce presupune conceperea, editarea, compilarea şi execuția unui program sursă. Testarea din cadrul colocviului va conține întrebări punctuale legate de problemele prezentate în unitățile de învățare. Nota finală va fi media aritmetică a notelor obținute la cele două testări.

Unitatea de învățare nr. 1

ELEMENTE GENERALE ALE LIMBAJULUI C

Cuprins **Pagina**

OBIECTIVELE unității de învățare nr. 1

Principalele obiective ale Unității de învățare nr. 1 sunt:

- Identificarea structurii şi a etapelor de execuție ale unui program C
- Cunoaşterea tipurilor de variabile ale limbajului C
- Utilizarea corectă a funcțiilor printf(), scanf() şi a secvențelor de evitare
- Utilizarea compilatorului Turbo C++ Lite pentru editarea compilarea şi execuția programelor C.

1.1 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().** Către această funcție principală sistemul de operare transferă controlul la lansarea în execuție a programului.

Exemplul următor conține o singură funcție, funcția **main().**

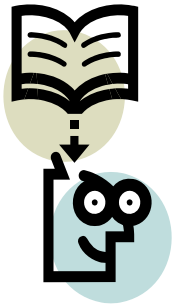
Ex. 1

```
#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 apărea 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.



De reținut !

Execuția unui program presupune parcurgerea etapelor de: ***compilare, link-editare*** şi ***de lansare*** în execuție.

Orice program sursă va conține obligatoriu funcția principală **main()**

1.2 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:

- char - caracter**,**

- **int** - întreg**,**

- **float** - real în virgula mobilă în simplă precizie**,**

- **double** - real în virgula mobilă în dublă precizie şi

- **void** - tip de variabilă neprecizat sau inexistent.

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) şi

- **short** (scurt).

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

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	$10^{-37} \div 10^{37}$ (6 digiti precizie)
Double	64	$10^{-308} \div 10^{308}$ (10 digiti precizie)
long double	80	$10^{-4932} \div 10^{4932}$ (15 digiti precizie)

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 afișa:

Evenimentul S are numărul 5 şi a avut loc la 11.300000

De reținut !

În limbajul C există cinci tipuri de variabile: **char, int, float, double** şi **void**. Primele 4 tipuri aritmetice de bază pot fi extinse cu ajutorul unor declarații suplimentare: **signed, unsigned**, **long** şi **short** . Fiecărui tip de variabilă îi corespunde un domeniu de valori şi o anumită dimensiune a zonei de memorie alocate în urma declarației.

1.3 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 afişării.

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 execuție ar fi apărut mesajul:

Evenimentul S are numarul 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 ("%6.2f\n", valoare);
printf (" %012f\n", valoare) ;
printf ("%10f\n", valoare);
}
```

În urma execuției 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 afișare umplerea cu zero a spațiilor goale.

1.4. 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 în 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.

1.5. 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.

Următorul program permite aflarea codului numeric al unei taste în zecimal, hexa si octal:

Ex. 5

```
#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();
}
```

De reținut !

În limbajul C afișarea unui mesaj sau a valorilor unor variabile pe ecran (în urma execuției programului) se poate efectua cu ajutorul funcției **printf()** şi a specificatorilor de format .

În mod similar, pentru introducerea datelor de la tastatură se poate utiliza funcția **scanf()**.

1.6. Compilatorul Turbo C++ Lite

Compilatorul Turbo C++ Lite, destinat mediului MS-DOS, dar care poate rula şi în Windows (în cadrul unei ferestre DOS), permite:

- crearea şi editarea programelor sursă în C şi C++,
- compilarea programelor editate,
- depanarea programelor şi
- rularea programelor create.

Lansarea în execuție a compilatorului se realizează cu ajutorul icon-ului corespunzător de pe desktop sau din Start/All Programs/ Turbo C++ Lite.

Ecranul de lucru albastru (fig.1.1) permite editarea programului sursă. Accesul la meniul din zona superioară se face cu ajutorul comenzii **F10** sau cu ajutorul mausului. Comenzile sunt disponibile prin selectarea din meniul compilatorului (fig.1.2), sau, în unele cazuri, cu ajutorul tastelor de comenzi rapide.

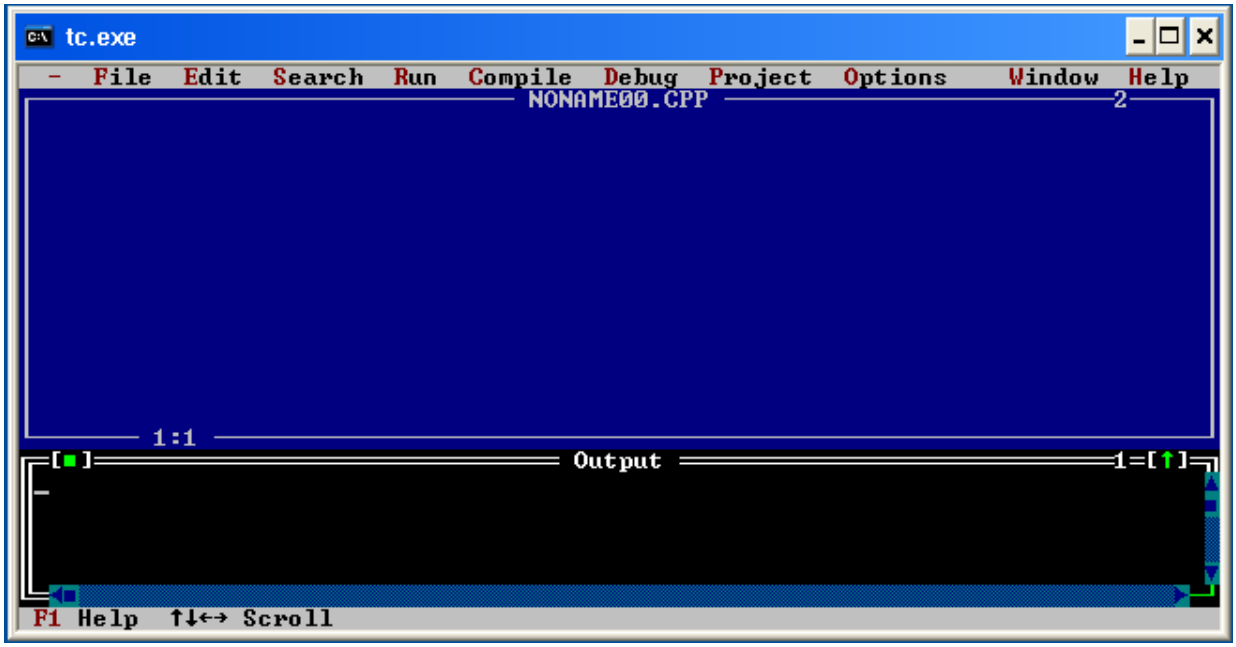


Fig.1.1 Configurația ecranului de lucru al Turbo C++ Lite

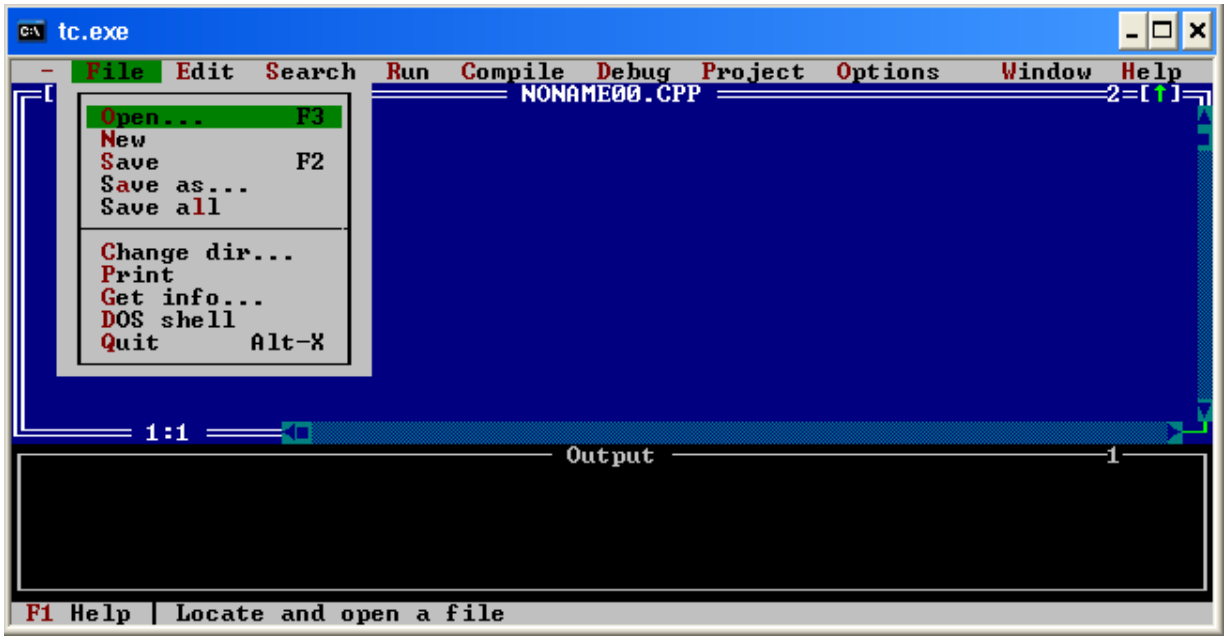


Fig.1.2 Meniul Turbo C++ Lite

Accesul la un set de comenzi din meniul Turbo C++ Lite se realizează prin combinații între tasta **Alt** şi prima litera (colorată în roşu) a numelui setului respectiv (fig.1.2):

File (Alt+F) – conține comenzi de creare a unui fişier sursă nou, deschiderea unuia existent, salvarea unui fişier, ieşirea din mediul DOS şi ieşirea din *Turbo C++ Lite;*

Edit (Alt+E) – conține comenzi de mutare (sau copiere) a unui bloc de text selectat în memoria Clipboard, de inserare a conținutului din Clipboard, anularea ultimei comenzi de editare, ştergerea unui bloc selectat;

Search (Alt+S) – conține comenzi pentru căutarea unui text, a unor declarații sau a pozițiilor unor erori în fişierele sursă;

Run (Alt+R) – conține comenzi pentru execuția programelor sursă, pentru marcarea începutului şi a sfârşitului sesiunilor de depanare;

Compile (Alt+C) – conține comenzi pentru compilarea programelor sursă, linkeditare şi crearea fişierului executabil;

Debug (Alt+D) – conține comenzi ce permit depanarea programelor sursă;

Project (Alt+P) – conține comenzi pentru realizarea unui proiect (program cu surse multiple): creare, deschidere, adăugare sau ştergere de fişiere, setare opțiuni;

Options (Alt+O) – conține comenzi ce permit modificarea de către utilizator a setărilor compilatorului;

Window (Alt+W) – conține comenzi ce permit gestionarea ferestrelor de lucru: deschidere, închidere, modificarea mărimii, a poziției etc;

Help (Alt+H) – conține componente ce permit cunoaşterea unor informații referitoare la mediul de programare, la noțiunile folosite, la funcțiile din bibliotecă şi la instrucțiunile limbajului.

Cele mai uzuale comenzi sunt:

- **Open (F2)** permite deschiderea unui fişier creat anterior;
- **New** permite deschiderea unui fişier nou;
- **Save (F2)** pentru salvare;
- **Compile (Alt+F9)** pentru compilare;
- **Run (Ctrl+F9)** pentru rulare;

- **User screen (Alt+F5)** permite accesul la ecranul în care sunt afişate rezultatele execuţiei programului (ecranul negru);
- **Alt+Enter** pentru maximizarea ecranului de lucru;
- **Ctrl+Break** pentru întreruperea execuţiei programului.

În cazul existenţei erorilor în programul sursă, în urma compilării vor apărea mesaje de eroare (fig.1.3) în partea de jos a compilatorului, în ecranul de culoare verde.

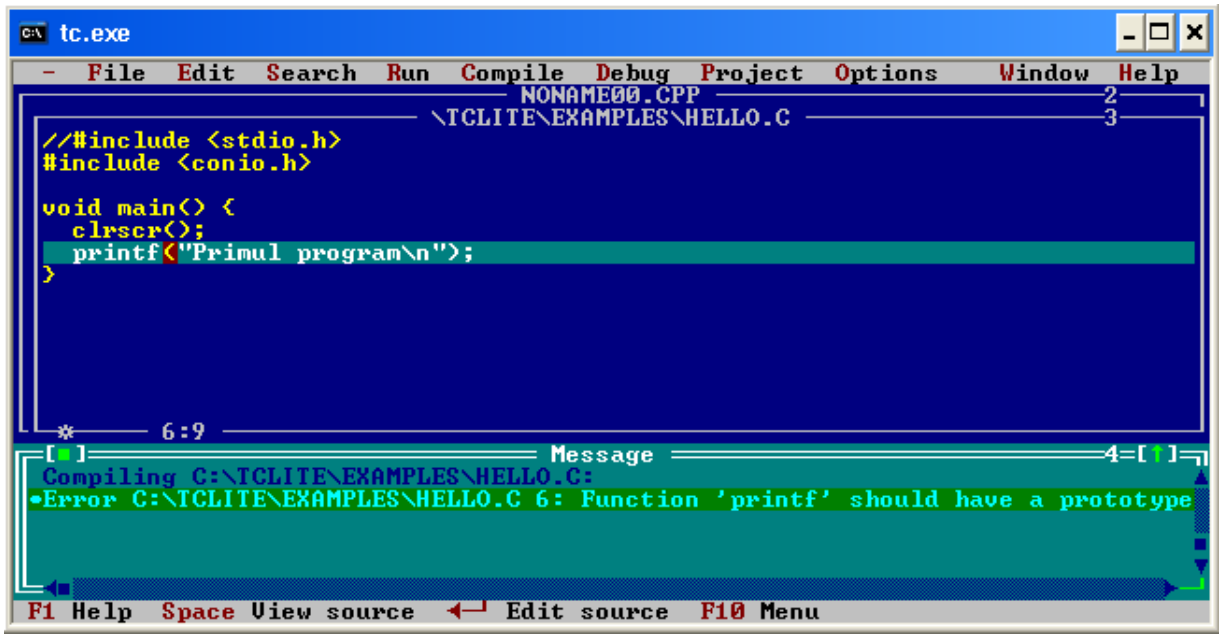


Fig.1.3 Mesaje de eroare

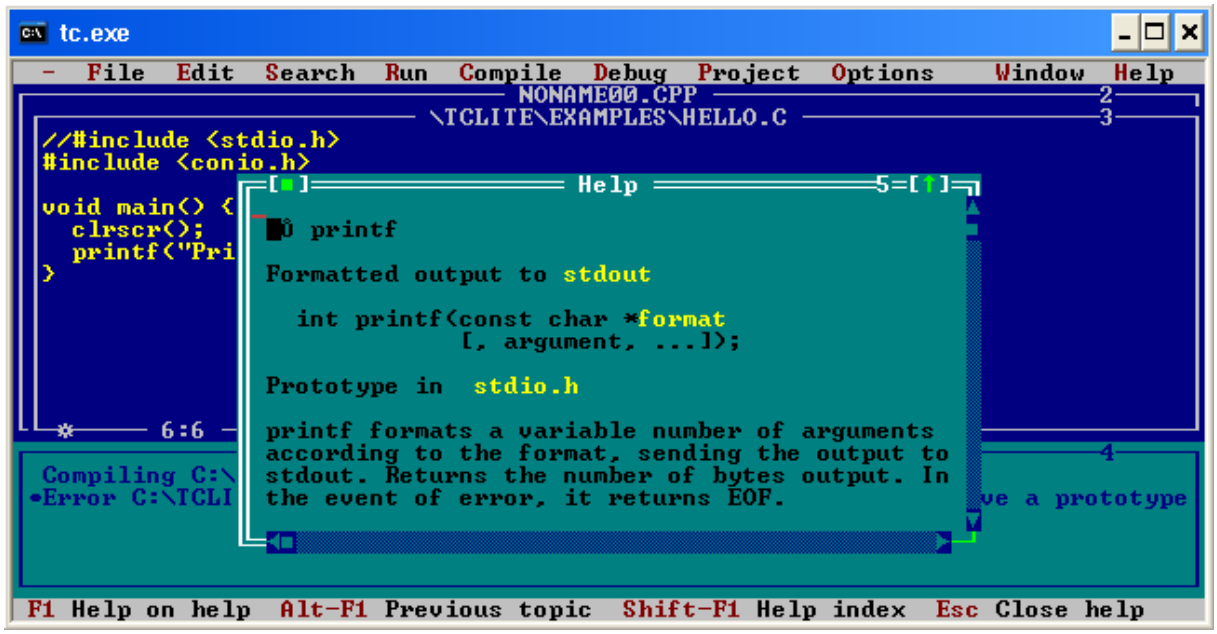
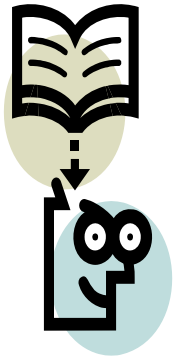


Fig.1.4 Meniul HELP

Poziționarea pe o anumită eroare în ecranul verde este corelată cu selecția unei anumite zone din ecranul albastru în care compilatorul a identificat eroarea. Astfel, eroarea este mai uşor detectată şi eliminată. Erorile se corectează în ordinea în care apar în mesaj.

După fiecare corecție, se recomandă compilarea, deoarece există posibilitatea ca o mare parte din următoarele erori să fie generate de prima eroare.

În etapa de corectare a erorilor sau în etapa de editare poate fi folosit meniul **Help** (fig.1.4) accesibil direct din meniul aflat în partea superioară sau prin intermediul tastelor de comenzi rapide. Poziționarea cursorului pe o anumită componentă a programului (de ex. funcție), urmată de succesiunea de taste **Ctrl+F1 (Topic search)** permite accesul la informațiile corespunzătoare din meniul Help.



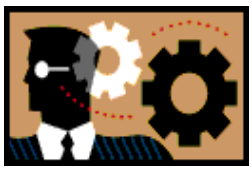
De reținut !

- Cu ajutorul compilatorului Turbo C++ Lite, un program sursă C poate fi editat, salvat (**F2**), compilat (**Alt+F9**) şi executat (**Ctrl+F9**).
- Rezultatele execuției programului pot fi vizualizate într-un ecran separat **User screen (Alt+F5).**

Lucrare de laborator

În cadrul lucrării de laborator aferentă noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1-5.

În acest scop, se va folosi compilatorul Turbo C++ Lite. Vor fi parcurse toate comenzile prezentate în secțiunea 1.6.



Test de autoevaluare 1.1

1. Ce tip poate fi folosit la declararea unei variabile corespunzătoare unui număr natural a cărui valoare poate fi maxim $6*10^4$?
2. Care este efectul utilizării specificatorului de format **%-5.1f** într-un apel al funcției **printf()** corespunzător afişării unei variabile de tip **float**?

Lucrare de verificare la Unitatea de învățare 1

Să se scrie un program sursă în limbajul C corespunzător rezolvării ecuației de gradul II cu coeficienți reali. Soluțiile ecuației se vor afişa (cu ajutorul funcției **printf()**) fiecare aliniate la stânga în câmpuri de 10 coloane, cu câte 3 cifre pentru partea zecimală.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. Se pot folosi tipurile **unsigned short int**, **long int**, **unsigned long int** etc.**.** Se preferă tipul **unsigned short int** deoarece necesită o zonă de memorie alocată mai mică decât în cazul celorlalte.
2. Variabila de tip **float** va fi afişată aliniată la stânga (semnul -) pe cinci coloane cu o singură cifră din partea zecimală.

Concluzii

Editarea oricărui program sursă în limbajul C trebuie realizată în conformitate cu regulile sintactice şi semantice aferente acestui limbaj. Declararea variabilelor trebuie realizată în funcție de tipul şi domeniul de valori al acestora, astfel încât spațiul ocupat de acestea în memorie să fie minim.

Apelul funcțiilor **printf()** şi **scanf()** impune utilizarea corectă a specificatorilor de format corelați cu tipul variabilelor şi modalitatea de afişare dorită.

Bibliografie

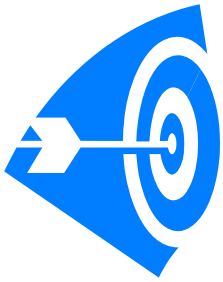
1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

Unitatea de învățare nr. 2

VARIABILE POINTER

OBIECTIVELE unității de învățare nr. 2

Principalele obiective ale Unității de învățare nr. 2 sunt:



- Declararea corectă a variabilelor pointer.
- Însuşirea regulilor specifice operațiilor cu pointeri.
- Utilizarea corectă a funcțiilor ce permit alocarea dinamică a memorie.

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 * defineşte 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:

```
……..
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 funcția **printf()** împreună cu specificatorul de format **%p**:

Ex.1:

```
#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 conține adresa obiectului: %d",*ptr);
*ptr=10;
printf("\nAcum, variabila var are valoarea %d\n",var);
}
```

În urma execuției programului se afişează:

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

În urma operației 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 condiții, 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**.

În situația în care, într-o operație de atribuire, tipurile variabilelor pointer diferă, pot apărea mesaje de eroare la compilare. Dacă:

…….

tip1 *ptr1;

tip2 *ptr2, var;

……..

în cazul atribuirilor:

ptr1=&var;

sau

ptr1=ptr2;

pot fi întâlnite situațiile:

- dacă **tip1** este **void**, atunci **tip2** poate fi oarecare;
- dacă **tip2** este **void**, atunci, în funcție de compilator, **tip1** poate fi oarecare (de ex. în **Turbo C** dar nu în Turbo C++)
- dacă **tip1** şi **tip2** sunt identice, evident atribuirea este corectă;
- dacă **tip1** şi **tip2** diferă, atunci compilatorul generează un mesaj de avertisment sau de eroare.

Pentru a evita aceste situații, în cazul în care tipurile diferă, se recomandă ca atribuirea să se facă printr-o conversie explicită folosind operatorul **cast**:

ptr1=(tip1*)&var;

sau

ptr1=(tip1*)ptr2;

pentru atribuirile prezentate anterior.

În exemplu următor, utilizarea operatorului **cast** elimină posibilitatea apariției mesajelor de avertisment sau de eroare din partea compilatorului:

……

```
int * ptr,a;
float v1=2.53,v2, *fptr;
void *vptr;
ptr=(int*)&v1;
v2=(float)*ptr;
v1=*(float*)vptr;
```

……

De reținut !

Tipul de bază este asociat rigid unui pointer, conform declarației. Totuşi, compilatorul C/C++ admite o serie de conversii prin care variabilele pointer pot fi folosite şi pentru a adresa obiecte de alte tipuri decât cel de bază, conversii valabile doar pentru operația curentă

Test de autoevaluare 2.1

1. În cadrul unui program sursă C există următoarele secvențe de instrucțiuni:

   a) **int m,n,*p;**
      m=3; n=9; p=n;

   b) **float y=2.1, z=3.14;**
      z=&y;

   c) **char c='A', **p, *q;**
      q = &c; p = &q;

   Identificați secvențele incorecte şi sursele erorilor.

2.2 Operații aritmetice cu pointeri

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

Operatorii relaționali permit compararea valorilor a doi pointeri:

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

În multe situații 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 operații de adunare sau de scădere între un pointer de obiecte şi un întreg. Deoarece un pointer este o valoare care indică o anumită locație din memorie, dacă adăugăm numărul 1 acestei valori, pointerul va indica următoarea locație din memorie. Deci, în cadrul acestor operații intervine şi tipul variabilei.

Regula după care se efectuează aceste operații este următoarea:

în cazul unei variabile pointer **ptr**:

tip *ptr;

operațiile aritmetice:

ptr+n şi ptr-n

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

n*sizeof(tip).

De exemplu:

Ex.2

```
……..
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--;                  /*  dp1<-- adresa_dp1-1*8   */
……..
```

În exemplul 2, sub forma de comentarii, au fost prezentate efectele operațiilor asupra variabilelor pointer. Aici, „**adresa_fp**” se referă la adresa conținută în variabila pointer **fp** şi nu la adresa lui **fp**.

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 diferența 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**.

Următoarele programe exemplifică regulile specifice operațiilor aritmetice cu pointeri:

Ex.3

```
#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);
}
```

Rezultatul execuției programului este:

fptr=7260, *fptr=7.3, &fptr=7258

Incrementare fptr:

fptr=7264, *fptr=0.0, &fptr=7258

iptr1=7268, *iptr1=5, iptr2=7266, *iptr2=10

Diferenta pointerilor iptr1 si iptr2 este=1

iptr1=7268, *iptr1=5, iptr2=7284, *iptr2=1

Se observă că în urma incrementării variabilei pointer **fptr** de tip **float** adresa conținută în acest pointer a crescut cu 4 (dimensiunea în octeți a lui **float**).

Diferența pointerilor **iptr1 si iptr2** este (7268-7266)/sizeof(**int**)=2/2=1

Efectul atribuirii: **iptr2=iptr1+8;** constă în creşterea cu 16 (8* sizeof(**int**)) a valorii variabilei **iptr2** față de **iptr1**.

Programul următor conține operații aritmetice pentru patru tipuri diferite de variabile pointer **int, float, double** şi **char**:

Ex.4

```
#include <stdio.h>
void main()
{
int i=12, *ip1,*ip2,*ip3,*ip4;
float f=2.173, *fp1,*fp2,*fp3,*fp4;
double d=-415.673, *dp1,*dp2,*dp3,*dp4;
char c='C', *cp1,*cp2,*cp3,*cp4;
ip2=&i;
ip1=ip2-1;
ip3=ip2+1;
ip4=ip2+2;
fp2=&f;
fp1=fp2-1;
fp3=fp2+1;
fp4=fp2+2;
```

```
dp2=&d;
dp1=dp2-1;
dp3=dp2+1;
dp4=dp2+2;
cp2=&c;
cp1=cp2-1;
cp3=cp2+1;
cp4=cp2+2;
printf("int i= %d  ip=&i \nip-1=%p  ip=%p  ip+1=%p  ip+2=%p\n\n",i,ip1,ip2,ip3,ip4);
printf("float f= %f  fp=&f \nfp-1=%p  fp=%p  fp+1=%p  fp+2=%p\n\n",f,fp1,fp2,fp3,fp4);
printf("double d= %le  dp=&d \ndp-1=%p  dp=%p  dp+1=%p
dp+2=%p\n\n",d,dp1,dp2,dp3,dp4);
printf("char c= %c  cp=&c \ncp-1=%p  cp=%p  cp+1=%p  cp+2=%p\n",c,cp1,cp2,cp3,cp4);
}
```

Rezultatul execuției programului este:

int i= 12 ip=&i
ip-1=1C68 ip=1C6A ip+1=1C6C ip+2=1C6E

float f= 2.17300 fp=&f
fp-1=1C5C fp=1C60 fp+1=1C64 fp+2=1C68

double d= -4.156730e+02 dp=&d
dp-1=1C4A dp=1C52 dp+1=1C5A dp+2=1C62

char c= C cp=&c
cp-1=1C48 cp=1C49 cp+1=1C4A cp+2=1C4B

Se observă că:

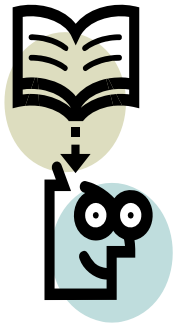
-pentru tipul **char** adresele sunt succesive

- pentru tipul **int** adresele cresc cu 2 unități (sizeof(**int**)=2)

- pentru tipul **float** adresele cresc cu 4 unități (sizeof(**float**)=4)

- pentru tipul **double** adresele cresc cu 8 unități (sizeof(**double**)=8)

Spre deosebire de ex.3, în ex.4 adresele conținute în variabilele pointer au fost afişate în hexazecimal prin utilizarea specificatorului **%p** în cadrul funcției **printf()**.

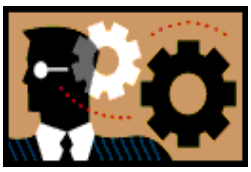


De reținut !

Operațiile aritmetice cu pointeri sunt supuse unor reguli şi restricții specifice.

Afişarea valorii variabilelor pointeri în hexazecimal se realizează cu funcția **printf()** împreună cu specificatorul de format **%p.**

Operația de scădere poate fi efectuată doar între doi pointeri de acelaşi tip.



Test de autoevaluare 2.2

2. În cadrul unui program sursă C există declarațiile:

   double *v1,*v2,d=34.123;

   int i;

   şi operațiile:

   v1=&d;

   v2=v1;

   v2++;

   i=v2-v1;

   Ce valoare va lua **i** în urma acestor operații?

   Dar dacă tipul este **int** in loc de **double**?

3. Care este rezultatul expresiilor ***(&var)** unde **var** este o variabilă de tip **float**?

   Dar al expresiei **&(*ptr)** unde **ptr** este un pointer de tip **int**?

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*.

În Turbo C++ zona de memorie în care se face alocarea se numeşte **heap**. Pentru modelul de memorie **small**, zona **heap** este cuprinsă între porțiunea ocupată de program şi variabilele permanente (care au dimensiunea fixă) şi cea ocupată de stivă (de dimensiune variabilă).

Față de alte limbaje, limbajul C nu posedă un sistem de alocare şi eliberare pentru variabile dinamice încorporat, din motive de reducere a complexității limbajului şi de creştere a flexibilității.

Dar, în biblioteca C, există un set de funcții destinate alocării şi eliberării memoriei în timpul execuției programului. Prototipurile acestor funcții 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**.

În exemplul următor este definită funcția **alocare()** ce permite introducerea de la tastatură a unui număr oarecare de valori reale, pentru care se alocă dinamic spațiu. În final este afişată adresa zonei de memorie alocate şi apoi memoria alocată este eliberată:

Ex.5

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
float *alocare(void)
        {
        float *fptr;
        int i, nr;
        printf("Dati numarul de valori de tip real:");
        scanf("%d",&nr);
        if(!(fptr=(float*)malloc(nr*sizeof(float))))
                {
                 puts("Memorie insuficienta");
```

```
                return NULL;
                }
        for (i=0; i<nr; i++)
                {
                printf("Introduceti valoarea %d: ",i+1);
                scanf("%f",fptr+i);
                }
        return fptr;
        }
void main(void)
        {
         float *fp;
         fp=alocare(); /* alocare pentru un numar nr de valori reale*/
         printf("%p",fp);
         free(fp);     /*eliberare memorie*/
        }
```

Pentru alocarea memoriei se poate folosi şi funcția:

void*calloc(unsigned nr_blocuri,unsigned mărime_bloc);

Spre deosebirea de **malloc()** modul de specificare a dimensiunii se face prin cei doi parametri, iar zona alocată este inițializată cu 0. În urma apelului funcției **malloc(),** se alocă o zonă cu dimensiunea:

(nr_blocuri*mărime_bloc) în octeți

Exemplul de la funcția **malloc()** se poate rescrie în felul următor:

```
float *fp;
fp=(float*)calloc(10,sizeof(float));
```

Eliberarea memoriei se face tot cu funcția **free()**.

În biblioteca C se găsesc şi alte funcții pentru alocarea şi eliberarea dinamică a memoriei: **coreleft(), realloc()** şi variantele pentru zona **heap** îndepărtată: **farmalloc(), farcalloc(), farfree()**,etc.

Următorul exemplu conține o secvență de alocări de valori de tip **float** care epuizează spațiului de memorie:

Ex.6

```
#include <stdio.h>
#include<alloc.h>
#include<process.h>
void main()
{
int i;
float *fp;
long m;
printf("Marime bloc= ");
scanf("%ld", &m);
for(i=1;;i++)
 {
  if(fp=(float*)calloc(m,sizeof(float)))
    {
     printf("Alocare bloc i=%d \n",i);

    }
  else
    {
     printf("Alocare imposibila\n");
     exit(1);
    }
 }
}
```

Programul afișează în urma execuției:

Marime bloc= 2000

Alocare bloc i=l

Alocare bloc i=2

Alocare.bloc i=3

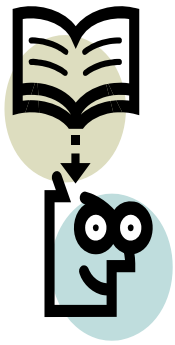
Alocare bloc i=4

Alocare bloc i=5

Alocare bloc i=6

Alocare imposibila

În ciclul **for** din exemplul precedent lipseşte condiție de testare (de ieşire din ciclu), ieşirea făcându-se cu funcția **exit()** în momentul în care nu mai este posibilă alocarea (variabila **fp** va avea valoarea **NULL**). Alocarea dinamică a memoriei a fost realizată prin apelul funcției **calloc()**.



De reținut !

Biblioteca limbajului C conține un set de funcții destinate alocării şi eliberării memoriei în timpul execuției programului.

Rezultatul apelului unei astfel de funcții este un pointer care conține adresa primului octet al zonei de memorie alocate.

Lucrare de laborator

În cadrul lucrării de laborator aferentă noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1, 3,4,5 şi 6.

Test de autoevaluare 2.3

4. Care este dimensiunea zonei de memorie alocată cu secvența:

double *dp;

dp=(double*)calloc(5,sizeof(double));

Lucrare de verificare la Unitatea de învățare 2

1. Să se scrie un program sursă în limbajul C care să conțină operațiile din exemplul 2. Pentru a verifica efectul acestor operații (trecut sub formă de comentariu în exemplu) programul va conține afișări ale valorilor variabilelor, utilizând funcția **printf().**
2. Să se scrie un program sursă în limbajul C care să conțină apelul unei funcții de alocare dinamică a memoriei pentru 10 valori de tip **double**. Zona de memorie alocată va fi inițializată cu 10 valori reale, se va face media aritmetică a acestora şi apoi se va elibera.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. a) Atribuirea **p=n** este incorectă deoarece **p** este pointer la **int** şi nu variabilă de tip **int**.
   b) Atribuirea **z=&y** este incorectă deoarece **z** este variabilă de tip **float** şi nu variabilă de tip pointer la **float** (nu poate conține adresa unei variabile **float**)
2. Valoarea va fi **1** indiferent de tipul celor doi pointeri.
3. Rezultatul este chiar variabila **var,** respectiv **ptr**, operatorii **&** şi ***** fiind complementari.
4. Se alocă spațiu pentru 5 variabile de tip **double**, deci 40 octeți.

Concluzii

Variabilele pointer sunt utilizate pentru a face referire la date cunoscute, prin adresele lor. Operațiile cu pointeri, de atribuire şi aritmetice sunt supuse unor reguli specifice.

Alocarea dinamică a memoriei, prin intermediul variabilelor dinamice, implică utilizarea variabilelor pointer, cu regulile aferente.

Bibliografie

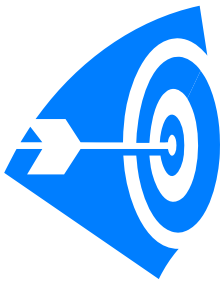
1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicatii, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

Unitatea de învățare nr. 3

TABLOURI

OBIECTIVELE unității de învățare nr. 3

Principalele obiective ale Unității de învățare nr. 3 sunt:



- Declararea corectă a tablourilor.
- Cunoaşterea funcțiilor dedicate şirurilor de caractere.
- Identificarea legăturilor dintre tablouri şi pointeri
- Definirea şi utilizarea corectă a funcțiilor în limbajul C

3.1 Declararea tablourilor

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**.

Programul următor încarcă tabloul **t1** cu numerele 1….10 şi apoi copiază conținutul lui **t1** în tabloul **t2:**

Ex.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 Şiruri de caractere

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),

- funcția **strlen(sir)** care returnează numărul de elemente al şirului **sir** sau

- funcția **strcmp(sir1,sir2)** care compară succesiv caracterele celor 2 şiruri. Dacă şirurile sunt identice returnează valoarea 0, iar dacă diferă, o valoare nenulă.

Utilizarea acestor funcţii este prezentată în exemplul următor, care testează introducerea parolei corecte *next*:

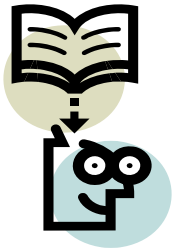
Ex.2

```
#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 declaraţie. În această situaţie, dimensiunea zonei de memorie alocate este fixată de compilator pe baza listei de constante utilizate la iniţializare.

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

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



De reţinut !

În limbajul C şirurile de caractere se declară ca tablouri unidimensionale de tip caracter (**char**).

3.3 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 execuţie, 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 declaraţiile:

```
……..
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.**

variabila	**ftab[0]**	**ftab[1]**	**ftab[2]**	**……..**	**ftab[i]**	**……**	**ftab[9]**
adresa	**&ftab[0]** ⇕ **ftab**	**&ftab[1]** ⇕ **ftab+1**	**&ftab[2]** ⇕ **ftab+2**	**……..**	**&ftab[i]** ⇕ **ftab+i**	**……**	**&ftab[9]** ⇕ **ftab+9**

Tab.3.1 Legătura dintre tablouri şi pointeri

Pe baza reprezentării din Tab.3.1 se observă următoarele echivalenţe:

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 echivalențe 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).

Aşadar, operația de adunare dintre pointeri şi întregi permite accesul la elementele unui tablou. La parcurgerea unui tablou, utilizarea pointerilor poate fi mai rapidă decât indexarea. Dacă dorim să determinăm lungimea unui şir de caractere, se poate folosi secvența:

…….
int contor;
char sir[10];
……
for(contor=0;str[contor];contor++);

sau, utilizând o variabilă pointer pentru parcurgerea şirului:

…….
int contor;
char sir[10],*p=sir;
……..
for(contor=0;*p;p++)contor++;
……..

În urma declarării unei variabile şir de caractere, compilatorul alocă zona de memorie necesară şi înscrie codurile ASCII ale caracterelor şi terminatorul ‘\0’. Adresa acestei zone de memorie poate fi atribuită unui variabile de tip pointer de caractere:

char*psir;
…….
psir=”un sir de caractere”;

Astfel, poate fi realizată declararea şi inițializarea unui şir sau pointer de caractere:

char* psir =” un sir de caractere”;

echivalentă cu:

char psir [20]=" un sir de caractere";

sau:

char psir []="un sir de caractere";

Pot fi utilizate şi tablouri care conţin pointeri la şiruri de caractere ca în exemplul:

……

char *sapt[]={"Luni","Marti","Miercuri","Joi","Vineri", "Sambata", "Duminica"};

……

unde tabloul **sapt** conţine zilele săptămânii.

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 matricei identificată prin **mat[0]**, iar **mat[0]** referă primul element al matricei **mat[0][0]**.

În general **mat[i]** conţine adresa elementului **mat[i][0],** iar **mat** poate fi văzut ca un tablou de 10 elemente **mat[0], mat[1]…mat[9]** (Tab.3.2)**.**

mat	→	mat[0]	→	mat[0][0]	mat[0][1]	. . . .	mat[0][9]
		mat[1]	→	mat[1][0]	mat[1][1]	. . . .	mat[1][9]
		. . . .	→	. . . .	. . . .	. . . .	. . . .
		mat[9]	→	mat[9][0]	mat[9][1]	. . . .	mat[9][9]

Tab.3.2 Legătura dintre tablourile bidimensionale şi pointeri

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)

Programul următor calculează urma unei matrice pătrate (suma elementelor de pe diagonala principală) utilizând variabile pointer pentru adresarea elementelor matricei. Aceste variabile pointer sunt inițializate, pentru fiecare linie, cu adresa de început a liniei respective.

Ex.3

```
#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 elementelor de pe diagonala principala este =%d\n",s);
}
```

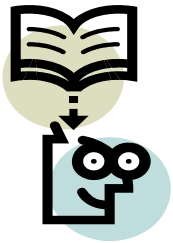
Rezultatul execuției programului este:

Dati dimensiunea matricei patrate:4
mat[1][1]=1
mat[1][2]=2
mat[1][3]=3
mat[1][4]=4
mat[2][1]=5
mat[2][2]=6
mat[2][3]=7
mat[2][4]=8
mat[3][1]=9
mat[3][2]=10
mat[3][3]=11
mat[3][4]=12
mat[4][1]=13
mat[4][2]=14
mat[4][3]=15
mat[4][4]=16
Suma elementelor de pe diagonala principala este =14

Programul următor 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.

Ex.4

```
#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);
}
```



De reținut !

Primul element al unui tablou are indicele [0]. Numele unui tablou fără index este un **pointer ce conține** adresa primului element din tablou.

Test de autoevaluare 3.1

1. În cadrul unui program sursă C există declarațiile:

   int itab[5], *iptr, i, ,j;

   double dmat[5][5];

   double *dp;

   şi operațiile:

   iptr=itab;

   dp=dmat[0];

   i=&itab[3]-iptr;

   j=dmat[1]-dp;

   a) Ce valori vor lua i şi j în urma acestor operații?

   b) Care este efectul operației de atribuire **dp=dmat**;.

3.4 Funcţii în limbajul C

În general, un program C este alcătuit din una sau mai multe funcţii. Întotdeauna există cel puţin o funcţie, funcţia **main()** care este apelată la lansarea în execuţie a programului.

Sintaxa generală a definirii unei funcţii este :

tip_fct nume_fct(listă_declaraţii_parametri)

{

<lista_declaraţii_locale>

listă_instrucţiuni

}

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

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

Ex.5:

```
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 **parametri formali** identificatorii din **lista_declarații_parametri** din definiția funcției şi **parametri efectivi** acele variabile, constante sau expresii din lista unui apel al funcției.

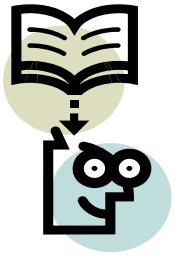
Se observă că parametri formali reprezintă variabilele locale ale funcției. Timpul lor de viață corespunde duratei de execuție a funcției.

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.5, atribuirea **a1=7.5** din finalul funcției nu modifică valoarea variabilei **r** din apelul **max(r,4.53);** .

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.



De reținut !

Parametri din definiția funcției se numesc **formali**, iar acele variabile, constante sau expresii din lista unui apel al funcției sunt **parametri efectivi.**

Limbajul C permite, în cazul funcțiilor, utilizarea pointerilor ca parametri sau ca rezultat returnat.

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.

Pentru o funcție ce schimbă valorile a două variabile între ele nu poate fi folosit transferul prin valoare ci doar cel prin referință:

Ex.6

```
#include <stdio.h>
void schimba(float *m, float* n)
{
  float temp;
  temp=*m;
  *m=*n;
  *n=temp;
}
main()
{
  float a,b;
  printf("Dati a= ");
  scanf("%f",&a);
  printf("Dati b= ");
  scanf("%f",&b);
  schimba(&a, &b);
  printf("Dupa schimbare a=%f si b=%f",a,b);
}
```

Rezultatul execuției programului este:

Dati a= 4.22
Dati b= 2.35
Dupa schimbare a=2.35 si b=4.22

Următorul program calculează suma elementelor unui vector utilizând o funcție care are printre parametri formali şi o variabilă pointer:

Ex.7

```
#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);
 }
```

Rezultatul execuției programului este:

Dati numarul de elemente al vectorului:5

Numar(1)=3.34

Numar(2)=4.23

Numar(3)=14.56

Numar(4)=23.34

Numar(5)=78.9

Suma elementelor vectorului este:124.37

Următorul program creează şi implementează o funcție care caută un caracter într-un şir şi returnează toate pozițiile pe care acesta este găsit. Pozițiile returnate sunt grupate într-un tablou, deoarece rezultatul funcției este de tip pointer la întreg.

Ex.8

```
#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);
}
```

Un pointer la o funcție este o variabilă care conține adresa codului executabil al funcției. Pointerii către funcții permit transferul ca parametru al adresei funcției şi apelul funcției cu ajutorul pointerului.

Sintaxa de declarare a unui pointer la o funcție este:

tip_fct (*var_point)(listă_declarații_parametri_formali);

unde:

var_point este un pointer de tipul "funcție cu rezultatul **tip_fct** şi parametri din **listă_declarații_parametri_formali**".

Variabila pointer ***var_point** permite apelul funcției a cărei adresă o conține astfel:

(*var_point) (listă_declarații_parametri_efectivi)

De exemplu, declarația:

double (*df) (double x, double y);

stabileşte că **df** este un pointer la o funcție care returnează o variabilă **double** şi are doi parametri **x** şi **y** de tip **double**.

Un exemplu de utilizare a pointerilor la funcții este crearea unui tablou de pointeri la funcții. Fiecare element din tablou pointează către funcții diferite, iar pentru a apela o anumită funcție trebuie indexat tabloul. Această implementare este mai eficientă decât folosirea instrucțiunii **switch ( )**.

Ex.9

```
# include <stdio.h>
# include <math.h>
double (* p[4]) (double a);
void main (void)
{
  int op;
  double a,rez;
```

```
  p[0] = sin; // adresa functiei sin ( )
  p[1] = cos; // adresa functiei cos ( )
  p[2] = tan; // adresa functiei tan ( )
  p[3] = atan; // adresa functiei atan( )
  printf ("Dati argumentul: ");
  scanf ("%le", &a);
  printf (" Optiuni: 0 - sin, 1 - cos, 2 - tan, 3 -atan  \n");
  do {
        printf (" Introduceti numãrul optiunii:");
        scanf ("%d", &op);
     } while (op < 0 || op >3);
  rez = (* p[op] ) (a);
  printf("Adresa functiei este: %p \n", p[op]);
  printf ("Rezultatul este: %le", rez);
}
```

Rezultatele execuției programului pentru patru opțiuni diferite este:

Dati argumentul: 0
Optiuni: 0 - sin, 1 - cos, 2 - tan, 3 -atan
Introduceti numãrul optiunii:0
Adresa functiei este: 0EC4
Rezultatul este: 0.000000e+00

Dati argumentul: 0
Optiuni: 0 - sin, 1 - cos, 2 - tan, 3 -atan
Introduceti numãrul optiunii:1
Adresa functiei este: 0FD8
Rezultatul este: 1.000000e+00

Dati argumentul: 1
Optiuni: 0 - sin, 1 - cos, 2 - tan, 3 -atan
Introduceti numãrul optiunii:2
Adresa functiei este: 0F04
Rezultatul este: 1.557408e+00

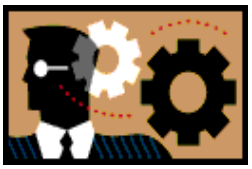
Dati argumentul: 1

Optiuni: 0 - sin, 1 - cos, 2 - tan, 3 -atan

Introduceti numãrul optiunii:3

Adresa functiei este: 0212

Rezultatul este: 7.853982e-01



Test de autoevaluare 3.2

2. În exemplul nr. 8, să se precizeze care sunt caracterul şi respectiv şirul în care se caută acest caracter. Sunt aceştia parametri formali sau parametri efectivi pentru funcția **find**.

Lucrare de laborator

În cadrul lucrărilor de laborator aferente noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1-7. De asemenea, vor fi concepute, editate, compilate şi executate probleme propuse ce impun cunoaşterea elementelor prezentate în această unitate de învățare .

Lucrare de verificare la Unitatea de învățare 3

1. Să se scrie un program sursă în limbajul C care să demonstreze echivalențele din paragraful **3.3**. Pentru afişarea valorilor variabilelor pointer se va folosi funcția **printf()** cu specificatorul de format **%p**.
2. Pornind de la exemplul 5, să se scrie un program sursă în limbajul C, care să calculeze media a trei numere reale cu ajutorul unei funcții.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. a) Valorile vor fi i=3 şi j=5.

   b) Eroare de compilare (tipuri diferite).

2. În definirea funcției **find**: **caracter** şi **sir** sunt parametri formali, iar în apelul acestei funcții (din corpul **main**) **'a'** şi **"acesta este sirul care va fi analizat"** sunt parametri efectivi.

Concluzii

În limbajul C există o strânsă legătură între 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.

Operația de adunare dintre pointeri şi întregi permite accesul la elementele unui tablou. La parcurgerea unui tablou, utilizarea pointerilor poate fi mai rapidă decât indexarea

Nu există un tip de date special pentru şiruri de caractere, acestea putând fi declarate ca tablouri unidimensionale de tip caracter (**char**).

Transmiterea parametrilor în urma unui apel al funcției poate fi realizat prin valoare sau prin referință. Pentru a modifica valoarea unei variabile indicate ca parametru efectiv într-o funcție, trebuie ca parametrul formal să fie de tip pointer, iar la apelare să i se ofere explicit adresa unei variabile. Acest procedeu se numeşte **transfer prin referință,** modificarea realizată de funcție asupra parametrilor efectivi fiind valabilă atât în interiorul cât şi în exteriorul funcției.

Bibliografie

1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Musatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

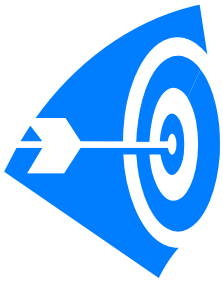
Unitatea de învățare nr. 4

STRUCTURI

Cuprins **Pagina**

OBIECTIVELE unității de învățare nr. 4

Principalele obiective ale Unității de învățare nr. 4 sunt:



- Declararea corectă a structurilor.
- Cunoaşterea accesului la elementele structurilor
- Manipularea variabilelor de tip pointer la structură
- Cunoaşterea variabilelor de tip uniune

4.1 Sintaxa de declarare a structurilor

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** ;

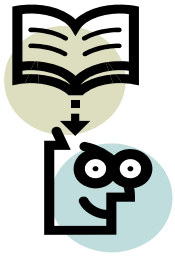
lista_var_struct = lista variabilelor de tip structură **nume_structura** definită anterior.

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.



De reținut !

Tipul **struct** permite gruparea sub un singur nume a mai multor variabile de tipuri diferite sau nu. Pot fi astfel reprezentate informații cu o structură complexă.

4.2 Accesul la elementele unei structuri

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};
```

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

Ex.1

```
#include<stdio.h>
struct student
{
        char nume[15];
        char prenume[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("    Prenume=");
         scanf("%s",stud[i].prenume);
         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)
            {
             printf("%s %s   %5.2f\n",stud[i].nume,stud[i].prenume,medie);
             }
    i++;
   }
}
```

Rezultatul execuției programului este:

Dati numarul de studenti: 3
NUME=Popescu
Prenume=Mihai
NOTA1=8
NOTA2=9
NOTA3=9
NOTA1=9
NUME=Dima
Prenume=Razvan
NOTA1=9
NOTA2=9
NOTA3=10
NOTA4=10
NUME=Petrescu
Prenume=Mircea
NOTA1=7
NOTA2=8
NOTA3=9
NOTA4=8

Popescu Mihai 8.75

Dima Razvan 9.50

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

Ex.2

```
#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++;
   }
}
```

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 punctului trei valoarea **20**.

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

Exemplul următor operează cu datele unei persoane: CNP, nume, data naşterii, localitatea de domiciliu şi salariul. Program determină vârsta persoanei prin aflarea datei sistemului (prin utilizarea **struct date**, o structură a sistemului de tip dată calendaristică şi a funcției **getdate**) şi afişează datele persoanei. Structura asociată datelor persoanei va avea un câmp de tip structură pentru data naşterii.

Ex.3

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
struct data
{unsigned zi,luna,an;};
struct persoane{
char cnp[13];
char nume[20];
char loc[30];
struct data dn;
int sal;
};
void main()
{ int Varsta;
struct persoane p;
struct date d;
clrscr();
printf("\n Dati CNP: ");
scanf("%s", p.cnp);
printf("\n Dati numele: ");
scanf("%s", p.nume);
printf("\n Dati localitatea: ");
scanf("%s", p.loc);
printf("\n Dati data nasterii in forma zz:ll:aaaa: ");
scanf("%2d:%2d:%4d",&p.dn.zi,&p.dn.luna,&p.dn.an);
printf("\n Dati salariul: ");
```

```
scanf("%d", &p.sal);
clrscr();
printf("Persoana %s este nascuta in data de %2d:%2d:%4d, locuieste in %s si are salariul
%d", p.nume,p.dn.zi,p.dn.luna,p.dn.an,p.loc,p.sal);
getdate(&d); //se afla data sistemului
Varsta=d.da_year-p.dn.an;
printf("\n Astazi suntem in %2d:%2d:%4d iar varsta este %d ani\n",
d.da_day,d.da_mon,d.da_year,Varsta);
getch();
}
```

Execuția programului afișează:

Dati CNP: 1234567891234

Dati numele: Popescu

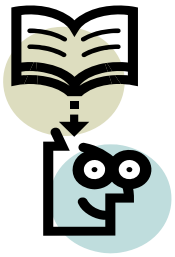
Dati localitatea: Craiova

Dati data nasterii in forma zz:ll:aaaa: 22:11:1967

Dati salariul: 1234

Persoana Popescu este nascuta in data de 22:11:1967, locuieste in Craiova si are salariul 1234

Astazi suntem in 22 : 12: 2008 iar varsta este 41 ani



De reținut !

Accesul la elementele unei structuri se face cu ajutorul operatorului punct „ **.** ” plasat între numele structurii şi numele membrului.

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.

4.3 Variabile pointer de tip structură

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):

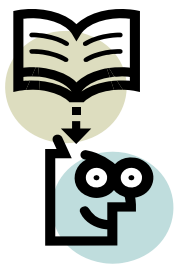
pct->y=12.3;

Exemplul 2 poate fi rescris, prin utilizarea variabilelor pointer astfel:

Ex.4

```
#include<stdio.h>
struct student
   {
        char nume[15];
        char prenume[15];
        int nota1;
        int nota2;
        int nota3;
        int nota4;
    } *stud;
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("    Prenume=");
            scanf("%s",(stud+i)->prenume);
            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)
          {
           printf("%s %s   %5.2f\n",(stud+i)->nume,(stud+i)->prenume,medie);
          }
     i++;
  }
}
```



De reținut !

În cazul variabilelor pointer de tip structură, pentru accesul la un element al structurii se utilizează **operatorul de selecție indirectă:'->'** (săgeata):

Limbajul C permite declararea funcţiilor cu argumente structuri sau chiar pointeri la structuri. Programul următor calculează suma şi produsul a două numere complexe. Numerele complexe sunt reprezentate cu ajutorul unei structuri cu două câmpuri, corespunzătoare părţii reale şi respectiv imaginare. Operaţiile de adunare şi înmulţire sunt realizate cu ajutorul a două funcţii ale căror parametri sunt pointeri la structura definită anterior:

Ex.5

```
#include<math.h>
#include<stdio.h>
struct complex
{float x;
 float y;
};
void adunare (complex *z1, complex *z2, complex *z3)
{z3->x=z1->x+z2->x;
 z3->y=z1->y+z2->y;
}
void inmultire (complex *z1, complex *z2, complex *z3)
{z3->x=z1->x*z2->x-z1->y*z2->y;
 z3->y=z1->x*z2->y+z1->y*z2->x;
}
void main(void)
{
struct complex z1,z2,z3;
printf("Dati partea reala a primului numar: ");
scanf("%f", &z1.x);
printf("Dati partea imaginara a primului numar: ");
scanf("%f", &z1.y);
printf("Dati partea reala a celui de-al doilea numar: ");
scanf("%f", &z2.x);
printf("Dati partea imaginara a celui de-al doilea numar: ");
scanf("%f", &z2.y);
adunare(&z1,&z2,&z3);
printf("Rezultatul adunarii este: %f + i*%f\n",z3.x,z3.y);
```

```
inmultire(&z1,&z2,&z3);
printf("Rezultatul inmultirii este: %f + i*%f",z3.x,z3.y);
}
```

Rezultatul execuției programului este:

Dati partea reala a primului numar: 1.2

Dati partea imaginara a primului numar: 3.4

Dati partea reala a celui de-al doilea numar: 7.5

Dati partea imaginara a celui de-al doilea numar: 4.5

Rezultatul adunarii este: 8.700000+i*7.900000

Rezultatul inmultirii este: -6.300000+i*30.900000

4.4 Uniuni

Variabilele de tip uniune sunt un caz special de structuri, care permit memorarea diferitelor tipuri de date în aceeaşi zonă de memorie.

Sintaxa generală de declarare a unei uniuni este similară cu a structurilor :

```
union    nume_uniune      {
        tip1    elem1;
        tip2    elem2;
        …
        tipn    elemn;
      } lista_var_uniune;
```

în care:

union = cuvânt cheie asociat uniunilor;

nume_uniune = numele dat de utilizator uniunii;

tipi = tipul elementului **elemi** ;

lista_var_uniune = lista variabilelor de tip uniune **nume_uniune** definită anterior.

Spațiul alocat în memorie pentru o variabilă de tip uniune corespunde tipului **tipi** cu dimensiune maximă. Dacă în cazul structurilor, spațiul alocat este egal cu suma spațiilor de memorie necesare tuturor câmpurilor, pentru uniuni se alocă spațiu doar pentru un element (cel de dimensiune maximă). Spre deosebire de structuri, la un moment dat, este accesibilă o singură componentă a unei variabile de tip uniune.

Diferit față de cazul structurilor, unde este posibilă o inițializare globală, variabilele de tip uniune pot fi inițializate doar pentru primul câmp. Toate celelalte operații definite în cazul structurilor pot fi utilizate şi în cazul uniunilor:

- referirea unui element prin utilizarea operatorului punct
- declararea variabilelor de tip pointer la uniuni şi referirea unui element cu ajutorul operatorului săgetă
- folosirea variabilelor de tip uniune ca parametri ai unei funcții etc.:

```
union u
        {
        int i;
        float f;
        }  u1;
int j = 7;
float r=45.89,m=9.8;
union u  *up=&u1;
union u  u2={73};
u1.i=j;
u1.f=r;
up->f=m;
```

Proprietățile variabilelor de tip uniune comparativ cu cele structură sunt evidențiate în exemplul următor:

Ex.6

```
#include <stdio.h>
struct i_si_f_si_d
{
 int i;
 float f;
```

```
 double d;
};
union i_sau_f_sau_d
{
   int i;
   float f;
   double d;
};
int main(void)
{
   struct i_si_f_si_d s;
   union i_sau_f_sau_d u;

   printf("dimensiune int: %d\n", sizeof(int));
   printf("dimensiune float: %d\n", sizeof(float));
   printf("dimensiune double: %d\n", sizeof(double));

   printf("dimensiune structura: %d\n",sizeof(struct i_si_f_si_d));
   printf("dimensiune uniune: %d\n",sizeof(union i_sau_f_sau_d));

   u.i = -11;
   printf("i=%d f=%lf d=%le \n", u.i, u.f,u.d);

   u.f = 7.12;
   printf("i=%d f=%lf d=%le \n", u.i, u.f,u.d);

   u.d =234.75;
   printf("i=%d f=%lf d=%le \n", u.i, u.f,u.d);
}
```

Rezultatul execuției programului este:

dimensiune int: 2

dimensiune float: 4

dimensiune double: 8

dimensiune structura: 14

dimensiune uniune: 8

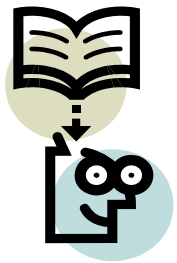
i=-11 f=0.000000 d=1.378104e-281

i=-10486 f=7.120000 d=1.378104e-281

i=0 f=0.000000 d=2.374500e+02

Se observă că dimensiunea variabilei de tip structură este **14=2+4+8** în timp ce dimensiunea variabilei de tip uniune este **8=max(2,4,8)**.

La un moment dat, pentru variabila uniune poate fi utilizat un singur câmp. De aceea inițializarea unui element al uniunii modifică automat valorile celorlalte câmpuri.



De reținut !

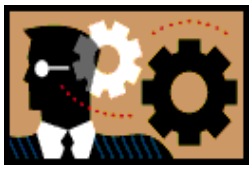
Variabilele de tip uniune se declară similar cu variabilele de tip structură.

Spațiul de memorie alocat pentru o variabilă de tip uniune este egal cu dimensiunea maximă a elementelor sale, spre deosebire de structuri unde spațiul alocat este egal cu suma spațiilor necesare tuturor câmpurilor .

Lucrare de laborator

În cadrul lucrărilor de laborator aferente noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1,2,3,4,5 şi 6.

De asemenea, vor fi concepute, editate, compilate şi executate probleme propuse ce impun cunoaşterea elementelor prezentate în această unitate de învățare .



Test de autoevaluare 4.1

1. În cadrul unui program sursă C există declarațiile:

   struct punct

   {int x;

   int y;

   } m,*n;

   Care din următoarele operații sunt greșit scrise:

   a) m.x=10;

   b) n.y=7;

   c) m->y=3;

   d) n->x=23;

Lucrare de verificare la Unitatea de învățare 4

1. Plecând de la exemplul 5, să se scrie un program sursă în limbajul C care să conțină două funcții pentru calculul diferenței şi a valorii absolute a două numere complexe.
2. Să se completeze exemplul 6 cu câte un câmp de tip şir de caractere pentru variabilele de tip structură şi uniune şi să se observe de o manieră similară diferențele între cele două implementări.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. b) Corect este: **n->y=7**

   c) Corect este: **m.y=3**

Concluzii

Limbajul C permite gruparea unor elemente, pentru a reprezenta date complexe, sub forma unor structuri.

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 tuturor membrilor. Pot fi declarate funcții cu argumente structuri sau pointeri la structuri.

O alternativă la variabilele structură o constituie, în anumite situații, utilizarea variabilelor de tip uniune declarate în mod similar şi ocupând o zonă de memorie redusă.

Bibliografie

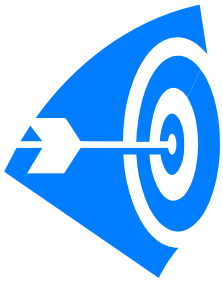
1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

Unitatea de învățare nr. 5

LISTE

OBIECTIVELE unității de învățare nr. 5

Principalele obiective ale Unității de învățare nr. 5 sunt:



- Implementarea listelor în limbajul C.
- Utilizarea listelor simplu înlănțuite
- Utilizarea listelor dublu înlănțuite

5.1 Noțiuni generale

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.

5.2. 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 (fig.5.1).

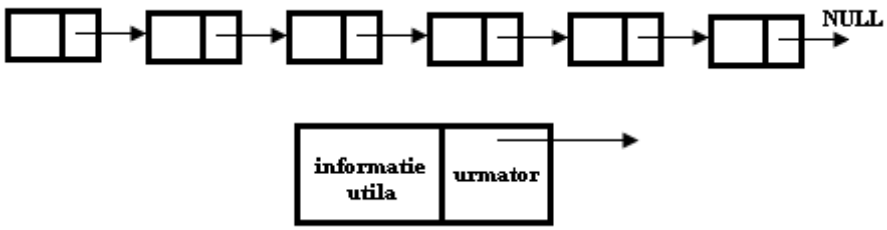


Fig.5.1 Liste simplu înlănțuite

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 importantă, 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:

1. se rup şi se refac legăturile şi
2. spațiul ocupat de acesta se eliberează.

De reținut !

Elementele unei liste simplu înlănțuite conțin un pointer de legătură ce memorează adresa nodului următor din listă

Următorul program creează o listă simplu înlănțuită a persoanelor dintr-o instituție în care se memorează numele, prenumele şi vârsta fiecăruia. Se afişează 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 afişează apoi lista persoanelor rămase.

Ex.1

```
#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);
}
```

În urma execuției programului se obține:

Numărul de persoane:5

Nume: Prenume: Popescu Mihai

Varsta:25

Nume: Prenume: Florea Cristina

Varsta:30

Nume: Prenume: Iovan Dragos

Varsta:42

Nume: Prenume: Pantea Ion

Varsta:66

Nume: Prenume: Ionescu Mihaela

Varsta:70

Lista persoanelor sub 40 de ani :

Popescu Mihai - 25

Florea Cristina - 30

Lista persoanelor ramase:

Popescu Mihai - 25

Florea Cristina - 30

Iovan Dragos - 42

Funcția **adauga()** din ex.1 realizează adăugarea unui element în listă şi conține cele trei etape:

1. ***alocare dinamică a spațiului pentru respectivul element cu secvența:***

struct nod *p;

p=(struct nod *)malloc(sizeof(struct nod)); *(alocare dinamică prin apelul funcției* ***malloc()*** *pentru un nod, cu utilizarea conversiei explicite de tip cast:* ***(struct nod *))***

if(p==NULL)

{ printf("Memorie insuficientă.\n"); return;}

(alocarea nu a reuşit pentru că memoria este insuficientă)

2. ***crearea elementului prin înscrierea informațiilor corespunzătoare:***

strcpy(p->nume,Nume);

strcpy(p->prenume,Prenume);

p->varsta=Varsta; *(se inițializează cele trei câmpuri ale nodului)*

3. ***legarea în listă:***

```
p->urm=NULL; (adăugarea se va face la sfârşitul listei, deci nu va exista un
                element următor lui p)
 if(prim==NULL) prim=ultim=p; (p este primul element introdus al listei)
   else
   { ultim->urm=p; (se leagă p cu elementul anterior)
     ultim=p;} (p devine ultimul element al listei)
```

Funcția **sterge()** din ex.1 realizează ştergerea unui element din listă şi conține cele două etape:

1. ***se rup şi se refac legăturile:***

```
if(!p) return; (dacă nu există în listă elementul p, ce se doreşte a fi şters, se iese din funcție)
  if(prim==p)
   {
    prim=p->urm;
    if(prim==NULL) ultim=NULL;
   } (dacă p este chiar primul element, atunci primul din listă va deveni cel care îi urmează )
     else
      { for(q=prim;q->urm!=p&&q->urm!=NULL;q=q->urm) (se parcurge întreaga
                                                      listă pentru a ajunge la elementul anterior lui p )
        if(q->urm==NULL)
        {
         printf("Elementul nu aparține listei.\n");
         return;
         }        (daca s-a parcurs lista şi nu s-a găsit elementul se iese din funcție)
        q->urm=p->urm;
                (se reface legătura între elementul anterior lui p şi cel următor lui)
        if(p==ultim) ultim=q; (daca p este ultimul element, atunci q, elementul său anterior
                               va deveni ultimul element din listă după ştergerea lui p)
      }
```

2. ***spațiul ocupat de acesta se eliberează:***

```
free(p);
```

5.3.Liste dublu înlănțuite

În funcția de ştergere a unui element dintr-o listă simplu înlănțuită este necesară găsirea elementului anterior celui care trebuie şters. De aceea, lista trebuie parcursă, de fiecare

dată, până se întâlneşte elementul anterior. Există situaţii în care cunoaşterea elementului anterior nu este posibilă şi de aceea ar fi util să putem determina rapid acest element. O soluţie ar fi folosirea listelor dublu înlănţuite.

Între nodurile unei liste dublu înlănţuite vor exista două relaţii 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 (fig.5.2).

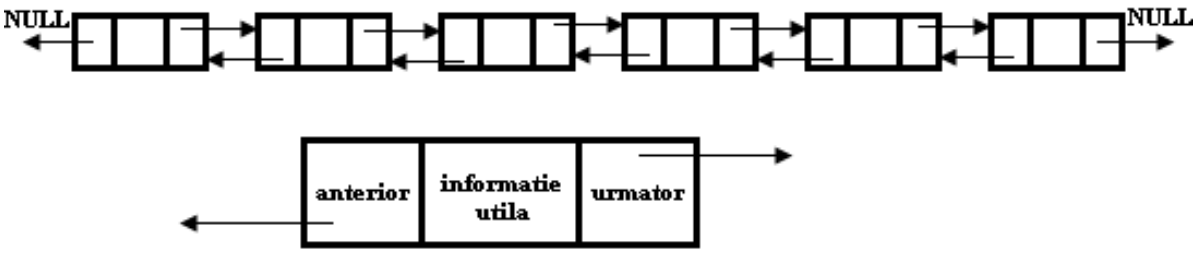
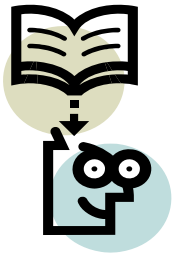


Fig.5.2 Liste dublu înlănţuite

Implementarea exemplului precedent se va face, în cazul utilizării listelor dublu înlănţuite, sub forma:

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



De reţinut !

Elementele unei liste dublu înlănţuite conţin doi pointeri de legătură ce memorează adresa nodului anterior şi respectiv a nodului următor din listă

Adăugarea unui element într-o listă dublu înlănțuită va necesita aceleaşi etape ca în cazul listelor simplu înlănțuite:

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

Astfel, ex.1 poate fi rescris în cazul utilizării listelor dublu înlănțuite astfel:

Ex.2

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <conio.h>
struct nod {
          char nume[15];
          char prenume[15];
             int varsta;
             struct nod *ant;
             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 insuficientã.\n");
     return;
      }
  strcpy(p->nume,Nume);
  strcpy(p->prenume,Prenume);
  p->varsta=Varsta;
  p->urm=NULL;
  ultim->urm=p;
  p->ant=ultim;
  ultim=p; }
```

```
void sterge(struct nod *p)
 {
 if(!p) return;
      p->ant->urm=p->urm;
      if(p==ultim) ultim=p->ant;
      else p->urm->ant=p->ant;
      free(p);
 }
void main(void) {
      char Nume[15],Prenume[15];
      int Varsta;
      int i,n;
      struct nod *p,*q;
      printf("Numãrul 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->urm;p!=NULL;p=p->urm)
          if(p->varsta <= 40) printf("%15s%15s - %d\n", p->nume,p->prenume,p->varsta);
      p=prim->urm;
       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->urm;p!=NULL;p=p->urm)
       printf("%15s%15s - %d\n",p->nume, p-> prenume, p->varsta);
}
```

Funcția **adauga()** din ex.2 diferă de cea din ex.1 doar în realizarea celei de-a treia etape:

3.legarea în listă:

p->urm=NULL; *(adăugarea se va face la sfârşitul listei, deci nu va exista un element următor lui p)*

ultim->urm=p; *(se leagă p cu elementul anterior prin câmpul **urm** al elementului ultim)*

p->ant=ultim; *(se leagă p cu elementul anterior prin câmpul **ant** al lui p)*

ultim=p; *(p devine ultimul element al listei)*

Funcția **sterge()** din ex.2 diferă de cea din ex.1 în realizarea primei etape:

1. se rup şi se refac legăturile:

if(!p) return; *(dacă nu există în listă elementul p, ce se doreşte a fi şters**,** se iese din funcție)*
p->ant->urm=p->urm; *(se reface legătura între elementul anterior lui p şi cel următor lui prin intermediul câmpul **urm** al elementului* p->ant*)*
if(p==ultim) ultim=p->ant; *(daca p este ultimul element, atunci elementul său anterior va deveni ultimul element din listă după ştergerea lui p)*
else p->urm->ant=p->ant; *(se reface legătura între elementul anterior lui p şi cel următor lui prin intermediul câmpul **ant** al elementului* p->urm*)*

5.4 Stive şi cozi

Pot fi întâlnite situații în care informația este reprezentată sub forma unei liste asupra căreia se impun anumite restricții legate de operațiile ce se pot executa asupra ei. Un exemplu de astfel de operații poate fi: adăugarea şi extragerea unui element.

O astfel de listă este cea de tip **stivă** care reprezintă o structură de tip LIFO (Last In First Out = ultimul intrat primul ieşit) în care toate adăugările (depunerile – în engleză **push**) şi ştergerile (sau extragerile - în engleză **pop**) sunt făcute doar la unul din capetele listei, numit vârful stivei. Listă de tip stivă constituie vagoanele din triaj garate pe o linie moartă (blocată la unul din capete) sau un set de farfurii aşezate una peste alta din care putem depune şi extrage doar în şi din vârful acestora.

Un alt exemplu de liste sunt cele de tip **cozi** care reprezintă structuri de tip **FIFO** (First In First Out = primul venit primul servit), în care toate adăugările (**put**) se fac la coada listei iar ştergerile (**get**) se fac doar la capătul listei. Listă de tip coadă întâlnim la o cale ferată pe un singur sens care trece printr-o staţie de încărcare a vagoanelor sau la un ghişeu oarecare la care primul venit este şi primul servit.

Exemplul următor simulează activitatea de stivuire a unor containere. Pentru fiecare container se va memora codul, conţinutul şi greutatea. Evenimentele legate de această activitate sunt „Asezare container in stiva” (tasta A) şi „Ridicare container din stiva” (tasta R). Pentru fiecare container stivuit se solicită codul, conţinutul şi greutatea, iar la plecare se vor afişa aceste informaţii.

Pentru implementarea stivei este utilizată o listă simplu înlănţuită, identificată prin variabila **varf (**elementul vizibil al listei**)**. Funcţia **push** returnează 0 în cazul în care nu se reuşeşte alocarea spaţiului necesar pentru noul element sau 1 în caz contrar. Funcţia **pop** oferă prin intermediul parametrilor săi de tip pointer, informaţiile referitoare la containerul scos din stivă. Funcţia returnează 1 dacă în stivă se găseşte cel puţin un container sau 0 în caz contrar.

Ex.3

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <conio.h>
struct stiva{
        int cod;
        char continut[20];
        int greutate;
        struct stiva * urm;
        };
```

```
struct  stiva *varf;
int push(int Cod,char *Continut, int Greutate)
      {
       struct stiva *p;
       p=(struct stiva *)malloc(sizeof(struct stiva));
       if(p==NULL) return 0;
       p->urm=varf;
       p->cod=Cod;
       strcpy(p->continut,Continut);
       p->greutate=Greutate;
       varf=p;
       return 1;
      }
int pop(int *pCod, char *pContinut, int *pGreutate)
     {
       struct stiva *p;
       if(!varf) return 0;
       p=varf;
      varf=p->urm;
      *pCod=p->cod;
       strcpy(pContinut,p->continut);
      *pGreutate=p->greutate;
      free(p);
      return 1;
     }
void main (void)
  {
  int Cod,Greutate;
  char Continut[20];
  int gata;
 char c;
 varf=NULL;
 gata=0;
 while(!gata)
```

```
        {
        clrscr();
        printf("Selectati operatia dorita:\n");
        printf("A- Asezare container in stiva\n");
        printf("R- Ridicare container din stiva\n");
        printf("T- Terminare program\n\n");
        c=getch();
        switch(c) {
                case 'A':       printf("Codul containerului:");scanf("%d",&Cod);
                                printf("Continutul containerului:");scanf("%s",Continut);
                                printf("Greutatea containerului:");scanf("%d",&Greutate);
                                if(push(Cod,Continut,Greutate))
                                printf("Containerul a fost stivuit ");
                                else printf("Memorie insuficientã");
                                break;
                case 'R':       if(pop(&Cod,Continut,&Greutate))
                                printf("A plecat containerul cu codul %d,  greutatea de %d tone
                                        continand %s",Cod,Greutate,Continut);
                                else printf("Nu mai exista nici un container");
                                break;
                case 'T':       gata=1;break;
                default:        printf("Operatie ilegala");break;
                }
   printf("\n\nApasati o tasta pentru a continua.\n");
  getch();
        }
}
```

În urma execuției programului sunt afişate:

Selectati operatia dorita:

A- Asezare container in stiva

R- Ridicare container din stiva

T- Terminare program

Se apasă tasta A:

Codul containerului:123

Continutul containerului: zahar

Greutatea containerului:20

Containerul a fost stivuit

Apasati o tasta pentru a continua.

Se apasă tasta A:

Codul containerului:45

Continutul containerului: faina

Greutatea containerului:23

Containerul a fost stivuit

Apasati o tasta pentru a continua

Se apasă tasta R:

A plecat containerul cu codul 45, greutatea de 23 tone continand faina

Apasati o tasta pentru a continua

Se apasă tasta R:

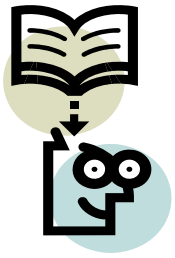
A plecat containerul cu codul 123, greutatea de 20 tone continand zahar

Apasati o tasta pentru a continua

Se apasă tasta R:

Nu mai exista nici un container

Apasati o tasta pentru a continua



De reținut !

Stivele sunt liste cu acces de tip **LIFO** (Last In First Out) în timp ce cozile sunt de tip **FIFO** (First In First Outt)

Lucrare de laborator

În cadrul lucrării de laborator aferentă noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1şi 2. Vor fi rezolvate problemele propuse.

Test de autoevaluare 5.1

1. Ce reprezintă, în cadrul funcției de adăugare a unui nod într-o listă:
   a) **sizeof(struct nod)**
   b) **(struct nod *)**
   c) **ultim->urm=p?**
2. Ce reprezintă, în cadrul funcției de ştergere a unui nod dintr-o listă:
   a) expresia de testare a instrucțiunii **for:**
   q->urm!=p&&q->urm!=NULL
   utilizată în cadrul funcției de ştergere a unui element dintr-o listă simplu înlănțuită,
   b) expresia **q=q->urm** a instrucțiunii **for?**

Lucrare de verificare la Unitatea de învățare 5

Să se scrie un program sursă în limbajul C în care să se creeze o listă dublu înlănțuită a studenților, în care să se memoreze numele, prenumele şi media notelor de la examene. Se afişează studenții bursieri (cu media mai mare de 8,50). Studenții cu media sub 6 vor fi eliminați din listă. Se vor afişa studenții rămaşi în listă.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. a) dimensiunea în octeți a zonei de memorie necesară unui nod;
   b) conversie explicită de tip recomandată în operațiile de atribuire între pointeri de tipuri diferite;
   c) legătura între ultimul element al listei şi elementul p nou adăugat la sfârşit.
2. a) Este condiția de ieşire din ciclu a instrucțiunii for. Ieşirea din ciclu se va realiza când s-a ajuns la elementul anterior celui care trebuie şters sau la sfârşitul listei.
   b) parcurgerea nod cu nod a listei.

Concluzii

Listele pot fi implementate în limbajul C cu ajutorul variabilelor dinamice de tip structură. Ordonarea elementelor unei liste se realizează prin folosirea variabilelor pointer care intră în componența elementelor. Utilizarea acestor variabile pointer dă un caracter recursiv elementelor listei, listele implementate astfel numindu-se *înlănțuite*.

Bibliografie

1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

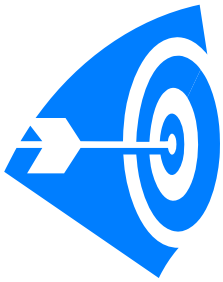
Unitatea de învățare nr. 6

FIŞIERE

Cuprins **Pagina**

OBIECTIVELE unității de învățare nr. 6

Principalele obiective ale Unității de învățare nr. 6 sunt:



- Înțelegerea noțiunilor legate de utilizarea fişierelor în limbajul C.
- Realizarea corectă a operațiilor de deschidere, citire, scriere si închidere a fisierelor.

6.1 Noțiuni generale

Un fişier reprezintă o colecție ordonată de înregistrări. Majoritatea operațiilor de intrare–ieşire se bazează pe manipularea fişierelor.

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

Într-un fişier text, toate datele sunt organizate ca şiruri de caractere, pe linii, separate între ele prin marcajul sfârşit de linie **'\n'** .

Operațiile specifice prelucrării fişierelor sunt:

- deschiderea unui fişier
- închiderea unui fişier
- crearea unui fişier
- citirea (consultarea) înregistrărilor dintr-un fişier
- actualizarea (sau modificarea) fişierului
- adăugare de înregistrări la sfârşitul fişierului
- poziționarea într-un fişier.
- ştergerea unui fişier
- schimbarea numelui unui fişier

În limbajul C, operațiile asupra fişierelor se realizează prin intermediul unor funcții din biblioteca standard (**stdio.h**). Transferurile dintre program şi dispozitivele periferice

(tastatură, monitor, disc, imprimantă, etc.) se realizează prin intermediul unor dispozitive logice predefinite numite **stream-uri** (fluxuri). Deci, un **flux de date** reprezintă totalitatea transferurilor de informaţii dintre un program şi un dispozitiv periferic. Astfel, datele introduse de la tastatura (intrarea standard) constituie un fişier de intrare, iar datele ce se afişează pe display (ieşirea standard) formează un fişier de ieşire.

Pentru fiecare fişier se asociază o structură de tip **FILE** predeclarată în **stdio.h**

Când se lansează în execuţie un program, în mod automat, se creează următoarele fluxuri de date, dispozitive logice predefinite (pointeri de tip **FILE**):

- **stdin** (standard input device) – asociat intrării standard (tastatură);
- **stdout** (standard output device) - asociat ieşirii standard (monitorul);
- **stderr** (standard error output device) - fişier care conţine mesajele de eroare;
- **stdaux** (standard auxiliary device) – asociat comunicaţiei seriale
- **stdprn** (standard printer) – asociat imprimantei.

Dintre acestea, **stdaux** şi **stdprn** sunt specifice mediului MS-DOS. La sfârşitul execuţiei programului, toate cele cinci fluxuri de date se vor închide automat.

6.2 Deschiderea şi închiderea fişierelor

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 structura 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 funcției **fopen** prin adăugarea literei **t** pentru text sau **b** pentru binar.

Dacă operația de deschidere are succes, funcția returnează un pointer la **FILE** (pointer care va fi folosit în continuare în operațiile 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 operații de actualizare şi cel de-al doilea de tip binar pentru scriere.

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

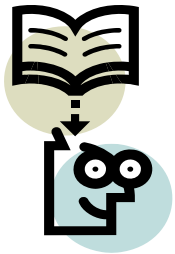
```
int fclose(FILE* fptr_fisier);
```

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

Funcția **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 secvența:

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



De reținut !

Apelul funcției **fopen()** generează un pointer la o structura de tip **FILE.** Prin intermediul celor doi parametri ai funcției se stabilesc numele fişierului care se va deschide sau crea şi respectiv modul în care este deschis fişierul.

În urma deschiderii unui fişier se stabileşte o legătură logică între fişier şi variabila pointer rezultată şi se alocă o zonă de memorie pentru realizarea mai eficientă a operațiilor de intrare / ieşire.

6.3 Scrierea şi citirea în/din fişier

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

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

care scrie în fişierul pointat de **fptr** datele cuprinse în **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 funcției **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 funcției **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 din **arg1, arg2,……argn,** care, de această dată, sunt pointeri.

Există şi funcții de citire şi scriere la nivel de bloc de date:

size_t fread(void *tab, size_t dim, size_t nr, FILE *fptr) – citeşte cel mult **nr** obiecte, de mărimea **dim,** din fişierul specificat de **fptr** şi le introduce în tabloul **tab**.

Funcția returnează numărul de elemente citite, care poate să fie mai mic decât **nr**.

size_t fwrite(const void *tab, size_t dim, size_t nr, FILE *fptr)- scrie **nr** obiecte de mărime **dim**, din tabloul **tab** în fişierul specificat de **fpoint**.

Funcţia returnează numărul de elemente scrise. În caz de eroare, numărul returnat este mai mic decât **nr**.

6.4. Poziţionarea în fişier

În afară de mecanismul de poziţionare implicit (inclus în operaţiile de citire şi scriere) pot fi folosite şi operaţiile de poziţionare explicită cu ajutorul funcţiilor:

a) long int ftell(FILE *fptr);

care întoarce, în caz de succes, poziţia curentă în fişier, exprimată prin numărul de octeţi faţă de începutul fişierului sau -1L în caz de eroare;

b) int fseek(FILE *fptr, long nr, int origine);

care modifică poziţia curentă în fişierul specificat de **fptr** cu **nr** octeţi relativ la al treilea parametru **origine astfel**:

- faţă de începutul fişierului, dacă **orig=0** (sau **SEEK_SET**)
- faţă de poziţia curentă, dacă **orig=1** (sau **SEEK_CUR**)
- faţă de sfârşitul fişierului, dacă **orig=2** (sau **SEEK_END**)
- fseek() întoarce rezultatul 0 pentru o poziţionare corectă, şi diferit de 0 în caz de eroare;

c) void rewind(FILE *fptr) poziţionează indicatorul fişierului la începutul acestuia;

d) int fgetpos(FILE *fptr, fops_t *poz) - înscrie în **poz** poziţia curentă din fişier şi întoarce zero în caz de succes;

O altă funcţie utilă în operaţiile cu fişiere este:

int feof(FILE *fptr);

care întoarce o valoare diferită de 0, dacă s-a detectat marcajul de sfârşit de fişier sau 0 în celelalte cazuri.

6.5 Exemple

Următorul program conține:

- o funcție de creare a unui fişier text ce conține modele de autoturisme.
- o funcție de actualizare prin adăugare
- o funcție de listare a conținutului fişierului

Ex.1

```
#include<stdio.h>
#include<process.h>
#include<conio.h>
#define fis "auto.txt"
FILE *fp;

void creare(char* nume_fis)     //crează fişierul text auto.txt
{
if((fp=fopen(nume_fis,"wt"))==NULL)
{
printf ("Eroare de creare");   //creare nereuşită
exit(1);
}
fclose(fp);
}
void adaug(char *nume_fis)       //adaugă model în fişierul text
{
char model[20],c;
clrscr();
if((fp=fopen(nume_fis,"a"))==NULL)
        {
        printf("Eroare de deschidere");
        exit(1);
        }c='d';
```

```
do
 {
printf("\n   Introduceti modelul de  autoturism: ");
scanf ("%s",model);
fprintf(fp,"%s\n",model);    // scrie în fisier
printf("\n    Mai introduceți alt model? d/n  ");
c=getch();
}         while(c=='d');
fclose(fp);
}

void listare (char* nume_fis)     //listează conținutul fişierului
{
char model[20];
if((fp=fopen(nume_fis,"rt"))==NULL)
          {
          printf("Eroare de deschidere");
          exit(1);
          }
while(!feof(fp))
          {
          if(fscanf(fp,"%s",model)==EOF) break;
          printf("\n%s",model);
          }
}

void main()
{
creare(fis);
adaug(fis);
listare(fis);
}
```

Un rezultat al execuției programului este afişarea:

Introduceti modelul de autoturism: Logan

Mai introduceți alt model? d/n

Introduceti modelul de autoturism: Fiesta

Mai introduceți alt model? d/n

Introduceti modelul de autoturism: Corsa

Mai introduceți alt model? d/n

Logan

Fiesta

Corsa

În afară de rezultatele afişate, pe unitatea de disc, se creeaza fisierul **auto.txt** ce conține datele introduse.

Următorul program creează un fişier text ce conține informații despre produsele dintr-un magazin. Urmează apoi o funcție de adăugare şi una de căutare în fişier după nume şi modificarea numărului de bucăți. În final, se listează conținutul fişierului modificat.

Ex.2

```
#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 bucati",s.nr);
        printf("\n%5.2f lei",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 doriți 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();
}
```

Rezultatul execuției programului este:

Nume:laptop

Numarul de bucati: 23

Pret: 1700

Mai doriți adăugare? (d/n):d

Nume:monitor

Numarul de bucati: 12

Pret: 322

Mai doriți adăugare? (d/n):d

Nume:maus

Numarul de bucati: 33

Pret: 23.5

Mai doriți adăugare? (d/n):n

laptop

23 bucati

1700.00 lei

monitor

12 bucati

322.00 lei

Maus

33 bucati

23.50 lei

Dati numele dupa care se cauta: monitor

Dati noua cantitate:10

laptop

23 bucati

1700.00 lei

monitor

10 bucati

322.00 lei

Maus

33 bucati

23.50 lei

Lucrare de laborator

În cadrul lucrării de laborator aferentă noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1şi 2. Vor fi rezolvate problemele propuse.

Test de autoevaluare 6.1

1. Ce reprezintă
   a) **FILE**
   b) **fopen(”fis.bin”,”r+”);**
2. Ce este greşit în secvența:

```
FILE *fp1;
float f=2.7543, *fp2;
fp2=fopen(“fis.dat”,”w”);
fprintf(fp1,”%d,%c,%f”,i,c,f);
```

3. Ce reprezintă expresia:

```
if((fp=fopen(nume_fis,"a"))==NULL)
```

Lucrare de verificare la Unitatea de învățare 6

Să se scrie un program sursă în limbajul C în care să se creeze şi să se listeze un fişier binar cu studenți. Pentru fiecare student se memorează numele şi două note. Să se completeze programul cu o funcție de actualizare prin adăugare a acestui fişier.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. b) operaţia de deschidere a unui fişier text numit „**fis.bin**” pentru operaţii de citire şi scriere;

2. Ultimele două operaţii sunt incorecte deoarece:

- **fp2** este un pointer la **float** şi nu la **FILE;**
- **fp1** trebuia să fie iniţializat în urma unui apel al funcţiei **fopen()** înainte de utilizarea sa într-un apel al funcţiei **fprintf()** (pentru a putea efectua operaţii de citire scriere fişierul trebuie deschis în prealabil);

3. Este o combinaţie între:

- deschiderea unui fişier şi
- testarea realizării acestei operaţiei.

Concluzii

În limbajul C, cu ajutorul funcţiilor din biblioteca standard (**stdio.h**) pot fi realizate operaţii specifice prelucrării fişierelor: deschiderea, închiderea şi crearea unui fişier, citirea înregistrărilor dintr-un fişier, actualizarea fişierului, adăugarea de înregistrări la sfârşitul fişierului, poziţionarea într-un fişier, ştergerea sau schimbarea numelui unui fişier.

De asemenea datele introduse de la tastatură (intrarea standard) constituie un fişier de intrare, iar datele ce se afişează pe display (ieşirea standard) formează un fişier de ieşire.

Deschiderea unui fişier stabileşte o legătură logică între fişier şi variabila pointer rezultată.

Bibliografie

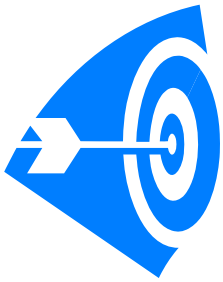
1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

Unitatea de învățare nr. 7

COMPLETĂRI ADUSE DE C++

OBIECTIVELE unității de învățare nr. 7

Principalele obiective ale Unității de învățare nr. 7 sunt:



- Cunoaşterea completărilor aduse de limbajul C++
- Utilizarea corectă a operațiilor de intrare ieşire cu **cin** şi **cout**
- Identificarea metodelor de alocarea dinamică a memoriei în C++

7.1. Noțiuni generale

C++ este un limbaj de programare dezvoltat în laboratoarele AT&T Bell de către Bjarne Stroustrup, plecând de la limbajul C (standardul ANSI C). Limbajul C++ poate fi considerat o extensie a lui C menită să-l perfecționeze şi să-l completeze cu caracteristici specifice programării orientate pe obiecte (POO). Astfel, C++ combină avantajele limbajului C (eficiență şi flexibilitate) cu cele oferite de tehnica POO, fără a impune însă utilizarea strictă a principiilor POO (pot fi scrise programe şi fără elemente POO).

În general, nu există diferențe între fişierele sursă scrise în C sau în C++, ambele acceptând directivele de compilator. Se poate utiliza extensia **cpp** pentru a diferenția fişierele sursă C++.

7.2. Operații de intrare/ieşire în C++

Limbajul C++, fiind o extensie a limbajului C, permite utilizarea tuturor funcțiilor de intrare/ieşire disponibile în C (de ex. **printf()** şi **scanf()**). În plus, pune la dispoziția utilizatorului, un sistem de intrare/ieşire mai flexibil şi mai comod.

Similar cu limbajul C, în C++ sunt predefinite dispozitive logice de intrare-ieşire:

- cin (console input) asociat intrării standard (tastatură) şi echivalent cu **stdin** din C;

- cout (console output) asociat ieşirii standard (monitorul) şi echivalent cu **stdout** din C.

Sintaxa de utilizarea pentru **cin** care permite citirea variabilei **var** este:

cin > > var;

În mod similar:

cout < < var;

permite afişarea pe monitor a variabilei **var.**

Operatorii: **>>** pentru intrare (utilizat împreună cu **cin)** şi **<<** pentru ieşire (utilizat împreună cu **cout)**, permit transferul informației cu formatare. Pot fi astfel transferate toate tipurile aritmetice precum şi şirurile de caractere. Cu toate că este posibil controlul formatului, acesta nu este obligatoriu, utilizarea acestor operații de intrare/ieşire fiind mai facilă decât în cazul funcțiilor **printf()** şi **scanf()** aşa cum reiese din echivalențele următoare:

int nr;

cin>>nr; echivalent cu **scanf("%d",&nr);**

cout<<nr; echivalent cu **printf("%d",nr);**

float m;

cin>>m; echivalent cu **scanf("%f",&m);**

cout<<m; echivalent cu **printf("%f",m);**

char c;

cin>>c; echivalent cu **scanf("%c",&c);**

cout<<c; echivalent cu **printf("%c",c);** echivalent cu **putch(c);**

char str[]="un text";

cout<<str; echivalent cu **printf("%s",str);** echivalent cu **puts(str);**

Pot fi realizate şi operații multiple:

cin > > varl > > var2... > > varN;

cout < < varl < < var2... < < varN;

Pentru a putea fi utilizate operațiile de intrare/ieşire C++ se impune includerea fişierului antet **iostream.h**.

Exemplul următor conține operații de intrare ieşire specifice limbajului C++:

Ex.l:

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

```
{
int nr;
char m;
char str[21];
cout << "Introduceți un număr intreg şi un caracter: ”;
cin >> nr >> m;
cout << "\n Ati introdus: "<< nr< < " şi " << m << '\n';
cout << "Introduceți numele dvs. si varsta:";
cin >> str>>nr;
cout << "Salut," << str << "!\n"<<”Aveti: ”<<nr<<” de ani!\n”; }
```

Rezultatul execuției programului este:

Introduceți un număr intreg si un caracter: 5 C

Ati introdus: 5 si C

Introduceți numele dvs: Mihai 23

Salut, Mihai!

Aveti 23 de ani!

7.3 Variabile referință

Limbajul C++ permite declararea unor identificatori ca referințe de obiecte (variabile sau constante). .

Declararea unei variabile referință se realizează cu simbolul &, folosind sintaxa:

tip & var_referinta = nume_obiect;

unde

- **tip** este tipul obiectului pentru care se declară referința, iar simbolul **&** precizează că
- **var_referinta** este numele unei variabile referință, în timp ce
- **nume_obiect** este obiectul a cărui adresă va fi conținută în **var_referinta**.

Exemplul următor conține declararea variabilei **vr** ca referință de obiecte **int**:

```
……
int i=5;
int *p=&i; //p este pointer ce conține adresa lui i
int& vr=i; // vr este referinta lui i (echivalent cu int &vr=i; sau int & vr=i;)
```

```
*p=35;     // echivalent cu i=35
vr=25;     // echivalent cu i=25
…….
```

Aici, **vr** şi **p** acționează asupra variabilei **i**, dar spre deosebire de pointerul **p** care poate primi adresa altei variabile de tip **int**, variabila referință **vr** este legată doar de variabila **i**.

Variabilele referință trebuie inițializate în momentul declarării şi nu li se pot modifica locațiile la care se referă.

În exemplul anterior a fost folosit delimitatorul // pentru adăugarea comentariilor de sfârşit de linie. Aceasta este o altă noutate a limbajul C++, tot textul care urmează după // până la sfârşitul liniei este considerat comentariu. Sunt admişi şi delimitatorii C: /* */ folosiți pentru comentarii ce se pot întinde pe mai multe linii.

Ca şi în cazul variabilelor pointer, utilizarea variabilelor referință permite modificarea obiectelor exterioare din interiorul unei funcții şi implementarea anumitor tehnici de programare.

În cazul realizării unei funcții ce schimbă valorile a două variabile între ele nu poate fi folosit transferul prin valoare ci doar cel prin referință. Varianta C necesită folosirea variabilelor de tip pointer, în timp ce în C++ pot fi folosite şi variabilele referință. Exemplul următor prezintă ambele variante:

Ex.2

a) varianta C

```
#include <stdio.h>
void schimba(float *m, float* n)
{
  float temp;
  temp=*m;
  *m=*n;
  *n=temp;
}
main()
{
  float a,b;
  printf("Dati a= ");
  scanf("%f",&a);
```

```
  printf("Dati b= ");
  scanf("%f",&b);
  schimba(&a, &b);
  printf("Dupa schimbare a=%f si b=%f",a,b);
}
```

b) varianta C+ +

```
#include <iostream.h>
void schimba(float &m, float &n)
{
float temp;
temp=m;
m=n;
n=temp;
}
main()
{
float a,b;
cout<<"Dati a= ";
cin>>a;
cout<<"Dati b= ";
cin>>b;
schimba(a, b);
cout<<"Dupa schimbare a="<<a<<" si b="<<b;
}
```

În ambele situații, rezultatul execuției va fi:

Dati a= 3.12
Dati b= 4.15
Dupa schimbare a=4.15 si b=3.12

Se observă o mai mare simplitate în cazul variantei C++ legată de scrierea funcției şi apelul său. În cazul variantei C++, declarațiile **float &m** şi **float &n** precizează că parametri

formali **m** şi **n** sunt referinţe la obiecte de tip **float**. În apelul funcţiei, parametri efectivi specifică obiectele care se asociază acestor referinţe.

7.4. Parametri cu valori implicite

Limbajul C++ permite declararea funcţiilor cu valori implicite ale parametrilor. În apelul unei astfel de funcţii pot fi omişi parametri efectivi corespunzători acelor parametri formali pentru care au fost declarate valori implicite. În acest caz se transferă automat valorile respective:

Ex.3:

```
#include <iostream.h>
void afiseaza_valori(int i=10, int j=20, int k=30)
{
  cout << "Parametri au valorile: i= "<< i;
  cout << "  j= " << j;
  cout << "  k= " << k << "\n";
}
void main()
{
  afiseaza_valori(5,99,45);       // apel normal
  afiseaza_valori(100,200);       // apel cu doua argumente
  afiseaza_valori(1000);          // apel cu un singur argument
  afiseaza_valori();              // apel fara argumente
}
```

Programul afişează:

Parametri au valorile: i= 5 j= 99 k=45

Parametri au valorile: i= 100 j= 200 k=30

Parametri au valorile: i= 1000 j= 20 k=30

Parametri au valorile: i= 10 j= 20 k=30

Utilizarea funcţiilor cu parametri impliciţi impune respectarea următoarelor reguli:

- valorile implicite pot fi specificate o singură dată, fie în prototip fie în declararea funcției.

- în declararea funcției, argumentele cu valori implicite trebuie plasate la sfârşitul listei.

7.5 Supradefinirea funcțiilor

În limbajul C++ pot fi supradefinite (supraîncărcate) funcțiile şi operatorii. Supradefinirea reprezintă facilitatea de a atribui unui simbol mai multe semnificații, ce pot fi deduse în funcție de context.

Astfel, pot exista mai multe funcții cu acelaşi nume, dar care lucrează cu tipuri de date diferite. De exemplu:

int fct(int a);

float fct(float m);

double fct(double d);

În cazul unui apel al unei funcții supraîncărcate, compilatorul alege care funcție va fi apelată după examinarea numărului şi a tipului argumentelor:

Ex.4

```
#include <iostream.h>
void functie(int a)
{ cout << a << '\n'; }
 void functie()
{ cout << "Nu are parametri \n"; }
 void functie(double a)
{ cout << a << '\n'; }
 void functie(int a, double b)
{ cout << a << ' ' << b << '\n'; }
 void main(void)
{
   functie(10);
   functie();
   functie(1.41);
```

```
    functie(10, 3.71);
}
```

Rezultatul execuției programului este:

10

Nu are parametri

1.41

10 3.71

În acest exemplu există patru funcții diferite cu acelaşi nume. Totuşi, în funcție de tipul şi numărul parametrilor efectivi, compilatorul alege varianta corectă.

7.6 Alocarea dinamică a memoriei

Pentru alocarea dinamică a memoriei, în limbajul C++ pot fi folosite funcțiile din grupul **malloc()** şi **free()**. În afara acestor funcții, completările C++ oferă o metodă nouă pentru gestiunea dinamică a memoriei prin utilizarea operatorilor **new** şi **delete**.

Sintaxa de alocarea dinamică a memoriei cu ajutorul operatorului unar **new**, este:

tip_ptr=new tip;

tip_ptr=new tip(init);

tip_ptr=new tip[n];

unde:

- **tip** este tipul variabilei dinamice (poate fi un tip de date oarecare);
- **tip_ptr** este variabilă pointer de tipul **tip**;
- **init** este valoarea cu care poate fi inițializată variabila dinamică.

Ultima varianta permite alocarea memoriei pentru un tablou cu n elemente de tipul tip.

Ca şi în cazul apelului funcției **malloc()**, rezultatul utilizării operatorului **new** este:

- un pointer de tipul **tip** ce conține adresa zonei de memorie alocate, dacă alocarea a reuşit,;
- un pointer cu valoarea **NULL** în cazul în care memoria este insuficientă sau fragmentată.

Astfel pentru:

int *iptr;

alocarea cu operatorul **new**:

iptr new int;

este echivalentă cu apelul funcției **malloc()**:

iptr=(int*) malloc(sizeof(int))

În exemplul:

float *fptr1, *fptr2;

fptr1=new float(3.14);//variabila intreaga neinitializata

int fptrt2=new float[10][20]; // tablou bidimensional int

se alocă dinamic spațiu pentru:

- o valoare de tip **float** initializată cu valoarea 3.14 şi
- un tablou bidimensional (10 x 20) de elemente de tip **float**.

Următorul exemplu, similar cu cel din unitatea 2, conține o secvență de alocări de valori de tip **float** care epuizează spațiului de memorie:

Ex.5:

```
#include <iostream.h>
#include<process.h>
void main()
{
int i;
float *fp;
long m;
cout << "Marime bloc= ";
cin >> m;
for(i=1;;i++)
 {
  if(fp=new float[m])
    {
    cout << "Alocare bloc i=";
    cout << i << "\n";
   }
  else
   {
   cout << "Alocare imposibila\n";
   exit(1);
```

```
    }
  }
}
```

Programul afişează în urma execuţiei:

Marime bloc= 2000

Alocare bloc i=l

Alocare bloc i=2

Alocare.bloc i=3

Alocare bloc i=4

Alocare bloc i=5

Alocare bloc i=6

Alocare imposibila

În ciclul **for** din exemplul precedent lipseşte condiţia de testare (de ieşire din ciclu), ieşirea făcându-se cu funcţia **exit()** în momentul în care nu mai este posibilă alocarea (variabila **fp** va avea valoarea **NULL**).

O zonă de memorie alocată cu operatorul **new** se eliberează cu ajutorul operatorului **delete**, cu sintaxa:

delete tip_ptr;

unde **tip_ptr** este variabila pointer creată în urma unei alocări cu operatorul **new.**

În exemplul:

float *fptr1;

fptr1=new float(3.14);

zona din memorie se eliberează folosind construcţia:

delete fptr1

Cu ajutorul completărilor aduse de limbajul C++, primul exemplu din unitatea de învăţare nr.5 Liste, poate fi rescris:

Ex.6

```
#include <iostream.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=new struct nod;
  if(p==NULL)
     {
     cout<<"Memorie insuficientã.\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)
                   {
                   cout<<"Elementul nu apartine listei.\n";
                   return;
                   }
         q->urm=p->urm;
         if(p==ultim) ultim=q;
          }
 delete p;
}
void main(void) {
      char Nume[15],Prenume[15];
      int Varsta;
      int i,n;
      struct nod *p,*q;
      cout<<"Numãrul de persoane:";
      cin>>n;
      prim=ultim=NULL;
      for(i=0;i<n;i++)
     {
           cout<<"Nume:";cin>>Nume;
              cout<<"Prenume:";cin>>Prenume;
           cout<<"Varsta:";cin>>Varsta;
           adauga(Nume,Prenume,Varsta);
      }
      cout<<"Lista persoanelor sub 40 de ani :\n";
      for(p=prim;p!=NULL;p=p->urm)
         if(p->varsta <= 40) cout<<p->nume<<" "<<p->prenume<<"  "<<p->varsta<<" ani\n";
      p=prim;
      while(p!=NULL)
```

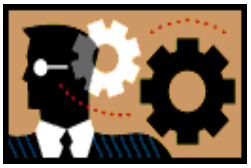
```
        {
         if(p->varsta > 65)
           {
              q=p->urm;
              sterge(p);
              p=q;
           }
         else p=p->urm;
        }
    cout<<"Lista persoanelor ramase:\n";
    for(p=prim;p!=NULL;p=p->urm)
        cout<<p->nume<<" "<<p->prenume<<"  "<<p->varsta<<" ani\n";
}
```

De reținut !

Pentru alocarea dinamică a memoriei, în limbajul C++ pot fi folosiți operatorii **new** şi **delete**.

Lucrare de laborator

În cadrul lucrării de laborator aferentă noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 1,2,3,4,5,6. Vor fi rezolvate problemele propuse.



Test de autoevaluare 7.1

1. În cazul unei variabile:
   char str[10];
   cu ce operație din C este echivalentă:
   cin>>str;
2. Cum poate fi rescrisă expresia:
   fptr=(float*) malloc(sizeof(float))
   utilizând completările aduse de C++

Lucrare de verificare la Unitatea de învățare 7

Să se scrie un program sursă în limbajul C++ echivalent exemplului 1 din unitatea 6 utilizând toate completările prezentate în această unitate.

Răspunsuri şi comentarii la întrebările din testele de autoevaluare

1. gets(str);
2. fptr=new float;

Concluzii

Limbajul C++ este o extensie a limbajului lui C pe care îl completează cu caracteristici specifice programării orientate pe obiecte (POO).

În C++ pot fi utilizate funcțiile de intrare/ieşire disponibile în C dar şi dispozitive logice de intrare-ieşire **cin** (console input) şi **cout** (console output).

Limbajul C++ permite declararea funcțiilor cu valori implicite ale parametrilor; apelul unei astfel de funcții poate omite parametri efectivi corespunzători acelor parametri formali pentru care au fost declarate valori implicite.

Pentru alocarea dinamică a memoriei, în limbajul C++ pot fi folosite funcțiile din grupul **malloc()** şi **free()** dar şi operatorii **new** şi **delete**.

Bibliografie

1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Muşatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro

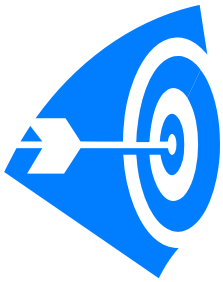
Unitatea de învățare nr. 8

REZOLVAREA PROBLEMELOR DE ANALIZĂ NUMERICĂ ÎN LIMBAJUL C++

Cuprins **Pagina**

OBIECTIVELE unității de învățare nr. 8

Principalele obiective ale Unității de învățare nr. 8 sunt:



Implementarea în limbajul C++ a metodelor numerice:

- de rezolvare a ecuațiilor şi a sistemelor de ecuații
- de interpolare
- de integrare a ecuațiilor diferențiale ordinare
- de calcul a vectorilor şi valorilor proprii

8.1. Rezolvarea numerică a ecuațiilor algebrice şi transcendente cu metoda bisecției

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 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 ξ=a;
2. f(b)=0 şi deci ξ=b;
3. f(a)·f(b)<0 şi atunci ξ aparține intervalului (a,b)
4. f(a)·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 $\mathbf{[a_i, b_i]}$ ca fiind jumătate din intervalul $\mathbf{[a_{i-1}, b_{i-1}]}$ prin metoda generală:

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

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

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

Se obțin astfel două şiruri convergente:

- şirul $\mathbf{a_n}$ al extremităților stângi ale intervalelor, care este monoton crescător:

$$\mathbf{a < a_1 < \ ... < a_n}$$

- şirul $\mathbf{b_n}$ al extremităților drepte ale intervalelor, care este monoton descrescător:

$$\mathbf{b > b_1 > \ ... > b_n}$$

Se observă şi că $\mathbf{b_n-a_n=(b-a)/2^n}$.

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

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

Ex. 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**.

Ex. 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.

Următorul program testează dacă ecuația: $\mathbf{x^3-9x^2+23x-15=0}$ are o rădăcină în intervalul **(-4.5,6)** şi află valoarea acesteia (în cazul în care există) cu o eroare de **0.000001**:

Ex.3

```
#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";
}
```

Rezultatul execuției programului este:

Functia are o radacina in intervalul (4.5,6) egala cu: 5

Următorul program găseşte soluția ecuației transcendente $x - e^{-x} = 0$ în intervalul (0.1,1), aplicând metoda bisecției cu o eroare de calcul de $\varepsilon = 0.0000000001$:

Ex.4

```
#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;
}
```

8.2 Interpolare

Interpolarea reprezintă o metodă numerică de aproximare a funcțiilor date sub formă tabelară. Având un set discret de date **$[x_i,y_i]$** (i=0,1,…,n) (obținut în urma unor experimente, măsurători etc.), metoda presupune găsirea unei funcții continue **f(x),** care să verifice **$y_i=f(x_i)$** (i=0,1,…,n). Funcția **f(x)** se numeşte funcție de interpolare, iar punctele **x_i** noduri ale rețelei de interpolare.

Uzual, funcția de interpolare are o formă simplă pentru a permite cu uşurință aflarea valorilor în orice punct al domeniului de definiție 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 funcției **f(x)** pentru valori ale lui **x** cuprinse între nodurile rețelei de interpolare se numeşte interpolare, iar dacă x se află în afara rețelei, extrapolare.

Cea mai utilizată funcție de interpolare este funcția 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:

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

condițiile din relația (1) sunt echivalente cu:

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

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 soluție unică pentru coeficienții $\mathbf{a_0, a_1 \ldots .a_n}$ şi deci pentru polinomul de interpolare.

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

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

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

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

Ex.5

```
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 rețeaua de interpolare are doar două puncte, interpolarea devine liniară:

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

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

În urma unor măsurători s-au obținut 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

Următorul program determină puncte între nodurile rețelei de interpolare (de exemplu pentru: x=0.35 ; x=0.64 ; x=0.89)

Ex.6

```
#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;
}
```

Rezultatul execuției programului pentru cele trei valori este:

Valoarea functiei de interpolare in punctul x=0.35 este: 20.803679

Valoarea functiei de interpolare in punctul x=0.64 este: 26.364653

Valoarea functiei de interpolare in punctul x=0.89 este: 26.864059

8.3. Integrarea ecuațiilor diferențiale ordinare

Se consideră problema Cauchy: **y'=f(x,y)**

cu condiția inițială: **$y(x_0)=y_0$**

Există mai multe metode numerice utilizate pentru rezolvarea unei astfel de ecuații diferențiale: metoda dezvoltării în serie Taylor, metoda lui Euler, metodele Runge-Kutta etc. Metodele Runge-Kutta sunt metode directe bazate pe dezvoltarea în serie Taylor şi necesită doar evaluarea funcției **f(x,y)**, nu şi a derivatelor acesteia.

Dintre metodele Runge-Kutta, cea de ordin 4 este mai utilizată deoarece este relativ simplă şi oferă un grad de precizie acceptabil. Această metodă este definită de relațiile:

$$x_{m+1} = x_m + h$$

$$y_{m+1} = y_m + \frac{h}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

cu:

$$k_1 = f(x_m, y_m)$$

$$k_2 = f\left(x_m + \frac{h}{2}, y_m + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(x_m + \frac{h}{2}, y_m + \frac{h}{2}k_2\right)$$

$$k_4 = f(x_m + h, y_m + hk_3)$$

În cadrul acestei metode, funcția este evaluată de patru ori, iar eroarea de trunchiere este de forma: $e_T \approx K \cdot h^5$

Exemplul următor foloseşte metoda Runge-Kutta de ordin 4, implementată în funcția RK4, pentru rezolvarea ecuației diferențiale **y'=x·y** cu condiția inițială **y(0)=1**. Deci **f(x,y)=x*y**, funcție notată în program cu **F**.

Ex.7

```
#include<math.h>
#include<iostream.h>
float F(float x,float y)
{
float val;
val=x*y;
return (val);
}
void RK4(float(*f)(float x,float y),
float x0,
float y0,
float h,
int nr,
float sol[])
{
int i;
float k1,k2,k3,k4;
sol[0]=y0;
for(i=1;i<=nr;i++)
{
```

```
k1=f(x0+(i-1)*h,sol[i-1]);
k2=f(x0+(i-1)*h+0.5*h,sol[i-1]+0.5*h*k1);
k3=f(x0+(i-1)*h+0.5*h,sol[i-1]+0.5*h*k2);
k4=f(x0+i*h,sol[i-1]+h*k3);
sol[i]=sol[i-1]+h*(k1+2*k2+2*k3+k4)/6.0;
cout<<"\n"<<(i)*h<<"     "<<sol[i];
}
}
void main(void)
{
float x0=0,y0=1,h=0.1,sol[10];
cout<<"\n0     1,000000";
RK4(F,x0,y0,h,10,sol);
}
```

Pentru un pas h=0,1 şi 10 valori, rezultatul execuție programului este:

0 1,000000
0,1 1,005013
0,2 1,020201
0,3 1,046028
0,4 1,083287
0,5 1,133149
0,6 1,197217
0,7 1,277621
0,8 1,377128
0,9 1,499302
1 1,648721

Programul permite modificarea din funcția principală a condiției inițiale şi a pasului h.

De reținut !

Utilizarea metodelor Runge-Kutta necesită doar evaluarea funcției **f(x,y)**, nu şi a derivatelor acesteia.

8.4 Sisteme de ecuații

Se consideră sistemul liniar de **n** ecuații cu **n** necunoscute:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ ---------- \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases}$$

sau matriceal **AX=B** unde:

$$A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ . & . & ... & . \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}; \qquad X = \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{pmatrix} \quad \text{si} \quad B = \begin{pmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{pmatrix}$$

Una dintre cele mai vechi metode iterative de rezolvare a sistemelor de ecuații liniare este metoda lui Jacobi. Aceasta presupune explicitarea fiecărei necunoscute din sistem, de pe diagonala principală, în funcție de celelalte necunoscute ale sistemului:

$$x_1 = \frac{b_1}{a_{11}} - 0 - \frac{a_{12}}{a_{11}}x_2 - \frac{a_{13}}{a_{11}}x_3 - ... - \frac{a_{1n}}{a_{11}}x_n$$

$$x_2 = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1 - 0 - \frac{a_{23}}{a_{22}}x_3 ... - \frac{a_{2n}}{a_{22}}x_n$$

$$----------$$

$$x_n = \frac{b_n}{a_{nn}} - \frac{a_{n1}}{a_{nn}}x_1 - ... - \frac{a_{nn-1}}{a_{nn}}x_{n-1} + 0$$

Se consideră o primă aproximare a soluțiilor sistemului oarecare:

$$x_1^{(0)}, x_2^{(0)}, ..., x_n^{(0)}$$

(de exemplu toate zero sau coloana termenilor liberi) şi se construiesc şirurile:

$$x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)}$$

$$x_1^{(2)}, x_2^{(2)}, ..., x_n^{(2)}$$

$$.....$$

$$x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}$$

pe baza condiției de recurență:

$$X^{(k)} = CX^{(k-1)} + D$$

unde:

$$X^k = \begin{pmatrix} x_1^k \\ x_2^k \\ \dots \\ x_n^k \end{pmatrix}; \quad C = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & -\frac{a_{13}}{a_{11}} & \dots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & -\frac{a_{23}}{a_{22}} & \dots & -\frac{a_{2n}}{a_{22}} \\ - & - & - & - & - \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & -\frac{a_{n3}}{a_{nn}} & \dots & 0 \end{pmatrix}; \quad D = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \dots \\ \frac{b_n}{a_{nn}} \end{pmatrix}$$

O condiție suficientă de convergență a metodei Jacobi este:

$$|a_{ii}| > \max_{j \neq i} |a_{ij}| \quad \text{pentru} \quad i = 1,2,\dots,n$$

Condiția de oprire a procesul iterativ este:

$$\max \left| x_i^k - x_i^{(k-1)} \right| < \varepsilon \quad \text{pentru} \quad i = 1,2,\dots,n$$

unde ε este eroarea maxim admisibilă.

În cazul în care a fost depăşit un număr maxim de iteraţii fără respectarea condiției anterioare, metoda nu este convergentă.

Metoda Jacobi prezentată mai sus este implementată în programul următor:

Ex.8

```
#include<stdlib.h>
#include<iostream.h>
#include<math.h>
void  Jacobi(float  a[20][20],float b[20],float x[20], int n,char *conv)
{
int   max=10000;
float eps=1.0e-6;
int    i,j,it;
float dif,eroare,s,xi[20];
for (i=0; i<n; i++)
xi[i]=0.0;
*conv='y';
i=0;
while (i<n)
```

```
        {
        if (a[i][i]==0.0)
                *conv='n';
        i = i+1;
        }
if (*conv=='y')
 {
 it=1;
 do
        {
        for (i=0; i<n; i++)
                {s = 0.0;
                for (j=0; j<n; j++)
                if (i!=j) s = s + a[i][j]*xi[j];
                x[i] = (b[i]-s)/a[i][i];
                }
        eroare =0.0;
        for (i=0; i<n; i++)
                { dif =  fabs(x[i]-xi[i]);
                if (fabs(dif)>1e20) it=max+1;
                if (dif>eroare)  eroare = dif;
        }
 for (i=0; i<n; i++) xi[i] = x[i];
 it = it+1;
 } while ((it<max)&&(eroare>eps));
 if (it>max)
        {
        *conv='n';
        cout<<"\n  Metoda nu este convergenta!";
        }
 else
  cout<<"Solutiile sistemului sunt:";
 for(i=0;i<n;i++)
 cout<<x[i]<<" ";
```

```
  }
}
void main (void)
{
 int i,j,n;
 char conv;
 float x[10];
 float a[10][20];
 float b[10];
 cout<<"Dati numarul de ecuatii: n=";
 cin>>n;
 cout<<"Dati elementele matricei A:";
 for (i=0;i<n;i++)
        for (j=0;j<n;j++)
        {cout<<"a["<<i<<","<<j<<"]=";
        cin>>a[i][j];
        }
for (i=0;i<n;i++)
        {
        cout<<"b["<<i<<"]=";
        cin>>b[i];
        }
 Jacobi(a,b,x,n,&conv);
}
```

Pentru sistemul de ecuații:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + x_4 = 12 \\ x_1 + 4x_2 = 2 \\ -x_1 + 4x_2 + 8x_3 = -2 \\ 2x_1 - x_2 - x_3 + 5x_4 = 4 \end{cases}$$

cu soluția exactă (2,0,0,0) rezultatul execuției programului este:

Dati elementele matricei A:

a[0,0]=6 a[0,1]=-2 a[0,2]=2 a[0,3]=1

a[1,0]=1 a[1,1]=4 a[1,2]=0 a[1,3]=0

a[2,0]=-1 a[2,1]=4 a[2,2]=8 a[2,3]=0

a[3,0]=2 a[3,1]=-1 a[3,2]=-1 a[3,3]=5

b[0]=12 b[1]=2 b[2]=-2 b[3]=4

Solutiile sistemului sunt: 2 -5.960464e-08 2.384186e-07 -9.536743 -08

8.5 Vectori şi valori proprii

Se consideră o matrice pătrată A de ordinul n cu elemente din corpul K (K=**R** sau K=**C**): Scalarul $\lambda \in K$ pentru care există un vector nenul $x^T = [x_1 \quad x_2 \quad x_3 \quad ... \quad x_n], \quad x \in K^n$ ce verifică **A·x=λ·x** se numeşte **valoare proprie** a matricei **A** iar vectorul **x** se numeşte **vector propriu** asociat valorii proprii **λ**.

Relația: **A·x=λ·x** poate fi scrisă matriceal:

$$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ - & - & - & - \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \lambda \cdot \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}$$

echivalentă cu:

$$\begin{pmatrix} a_{11}-\lambda & a_{12} & ... & a_{1n} \\ a_{21} & a_{22}-\lambda & ... & a_{2n} \\ - & - & - & - \\ a_{n1} & a_{n2} & ... & a_{nn}-\lambda \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = 0$$

care reprezintă un sistem omogen ce admite întotdeauna soluția banală:

$$x_1 = x_2 = x_3 = ... = x_n = 0.$$

În plus, sistemul liniar omogen admite soluții nenule dacă şi numai dacă:

det (A-λ·I_n)=0 unde I_n este matricea unitate de ordinul n.

Determinantul în necunoscuta λ reprezintă un polinom de grad n şi se numeşte **polinomul caracteristic** al matricei **A,** iar egalarea sa cu zero **ecuație caracteristică** a matricei **A**.

Orice matrice de ordinul n cu coeficienți complecşi are exact n valori proprii (nu neapărat distincte) care sunt rădăcinile ecuației caracteristice.

Calculul vectorilor şi a valorilor proprii simplifică operațiile cu matrice în rezolvarea sistemelor de ecuații diferențiale şi în alte operații mai complicate.

Cea mai simplă metodă de calcul a valorilor proprii şi ale vectorilor proprii asociate este **metoda puterii**.

Pentru matricea pătrată A de ordinul n se presupune existența a n valori proprii distincte $\lambda_1, \lambda_2, \ldots \lambda_n$, şi a n vectori proprii $x_1, x_2, \ldots x_n$ independenți. În aceste condiții, un vector oarecare $y \in K^n$ poate fi scris ca o combinație liniară de cei n vectori proprii ai matricei A:

$$y = a_1x_1 + a_2x_2 + ... + a_nx_n$$

Dacă înmulțim această egalitate cu matricea A rezultă:

$$Ay = Aa_1x_1 + Aa_2x_2 + ... + Aa_nx_n = a_1\lambda_1x_1 + a_2\lambda_2x_2 + ... + a_n\lambda_nx_n$$

iar dacă se continuă înmulțirea cu A a noilor egalități, după pasul k se obține :

$$A^ky = A^ka_1x_1 + A^ka_2x_2 + ... + A^ka_nx_n = a_1\lambda_1^kx_1 + a_2\lambda_2^kx_2 + ... + a_n\lambda_n^kx_n$$

care poate fi rescrisă :

$$A^ky = \lambda_1^k(a_1x_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^k x_2 + ... + a_n\left(\frac{\lambda_n}{\lambda_1}\right)^k x_n \approx a_1\lambda_1^kx_1$$

dacă se consideră λ_1 ca fiind cea mai mare valoare proprie şi k este mare.

Aceiaşi aproximare poate fi făcută şi pentru ecuația :

$$A^{k-1}y = \lambda_1^{k-1}(a_1x_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^{k-1} x_2 + ... + a_n\left(\frac{\lambda_n}{\lambda_1}\right)^{k-1} x_n \approx a_1\lambda_1^{k-1}x_1$$

Din împărțirea ultimelor două egalități se obține valoarea proprie de valoare maximă λ_1:

$$\lambda_1 \approx \frac{A^ky}{A^{k-1}y} = \frac{y_k}{y_{k-1}}$$

unde cu y_k s-a notat $A^k \cdot y$.

Se observă că $y_k \approx a_1\lambda_1^kx_1$ şi deci este un aproximant al vectorului propriu x_1 corespunzător variabilei proprii λ_1.

Deoarece y_k are componente mari în valoare absolută se consideră ca vector propriu corespunzător lui λ_1 vectorul $\frac{y_k}{\|y_k\|}$.

Această metodă este implementată în exemplul următor:

Ex.9

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

```
#include<stdlib.h>
int main()
{
 float a[20][20],x[20],y[20],val=0,temp;
 int n,i,j;
 clrscr();
 cout<<"Dati dimensiunea matricei: ";
 cin>>n;
 cout<<"\nDati elementele matricei \n ";
 for(i=0;i<n;i++)
    {
     for(j=0;j<n;j++)
       {
        cout<<"a["<<i+1<<"]["<<j+1<<"]=" ;
        cin>>a[i][j];
        }
     }
cout<<"Dati vectorul initial: \n ";
for(i=0;i<n;i++)
    {
    cout<<"x["<<i+1<<"]= ";
    cin>>x[i];
    }
do
  {
  for(i=0;i<n;i++)
    {
     y[i]=0;
     for(j=0;j<n;j++)
          y[i]+=a[i][j]*x[j];
     }
   for(i=0;i<n;i++)
          x[i]=y[i];
    temp=val;
```

```
val=0;
for(i=0;i<n;i++)
   {
    if(fabs(x[i])>fabs(val))
       val=x[i];
   }
for(i=0;i<n;i++)
     x[i]/=val;
}while(fabs(val-temp)>0.0001);
cout<<"Valoarea proprie este : "<<val<<endl;
cout<<"Vectorul propriu este: ";
for(i=0;i<n;i++)
   cout<<endl<<x[i];
   getch();
}
```

Pentru matricea: $\begin{pmatrix} 2 & 1 & 1 \\ -1 & 2 & -1 \\ 1 & -1 & 2 \end{pmatrix}$ şi vectorul inițial $(1\ 0\ 0)^T$ execuția programului este:

Dati dimensiunea matricei: 3

Dati elementele matricei

a[1][1]=2

a[1][2]=1

a[1][3]=1

a[2][1]=-1

a[2][2]=2

a[2][3]=-1

a[3][1]=1

a[3][2]=-1

a[3][3]=2

Dati vectorul inițial:

x[1]=1

x[2]=0

x[3]=0

Valoarea proprie este 3.000134

Vectorul propriu este:

-8.9124844e-05

1

-1

Se obține aşadar valoarea proprie 3 şi vectorul propriu asociat $(0\ 1\ -1)^T$

Lucrare de laborator

În cadrul lucrărilor de laborator aferente noțiunilor prezentate vor fi editate, compilate şi executate programele sursă prezentate în exemplele 3,4,6,7 şi 8. Vor fi rezolvate problemele propuse.

Test de autoevaluare 8.1

1. Care este utilitatea funcției matematice **poly()** şi unde se găseşte prototipul acesteia ?

Lucrare de verificare la Unitatea de învățare 8

1. Să se scrie un program sursă în limbajul C prin care să se verifice dacă ecuația $2x^3-3x^2+1=0$ are o rădăcină în intervalul (-1,0), iar în caz afirmativ să se găsească această soluție.

2. Să se scrie un program sursă în limbajul C prin care să se găsească soluțiile ecuației diferențiale $y'=x^2+y^2$ în punctele x=0.1; 0.2, 0.3; 0.4; 0.5, 0.6; 0.7, 0.8, 0.9 şi 1 ştiind că y(0)=1.

Concluzii

Limbajul C++ permite implementarea facilă a metodelor numerice de rezolvarea a ecuațiilor şi a sistemelor de ecuații, de interpolare, de integrare a ecuațiilor diferențiale ordinare şi de calcul a vectorilor şi valorilor proprii.

Bibliografie

1. Catrina, O., Cojocaru I., Turbo C++, Editura Teora, Bucureşti 1993
2. Lascu, M., Musatescu C., Marian Gh.: Limbajul C. Aplicații, Ed. Spirit Romanesc, Craiova,1997
3. Muşatescu C., Iordache S., Limbajul C, Reprografia Universității din Craiova, 1997
4. Limbaje de programare, lucrări de laborator, format electronic postat pe www.em.ucv.ro