

Adrian DEACONU

Programare procedurală

Cuvânt înainte

Cartea de față se dorește a fi, în principal, un ghid pentru studenții din domeniul Informatică, dar, evident, ea poate fi utilă tuturor celor care vor să învețe să programeze procedural în limbajul C.

La fiecare capitol, sunt date exemple sugestive, care ilustrează din punct de vedere practic elemente de noutate. Este bine ca aceste exemple să fie înțelese și, acolo unde este nevoie, să fie scrise și rulate de către cititor. Programele din această carte nu conțin erori, deoarece ele au fost întâi testate și abia apoi introduse în lucrare.

În general, tot ceea ce este prezentat în această carte (teorie și aplicații) este recunoscut atât de compilatorul C al firmei Borland, cât și de compilatorul Visual C al companiei Microsoft.

Autorul.

Câteva precizări înainte de a trece la studierea acestui curs

Cursul este structurat în 19 capitole. Este prezentat limbajul de programare C.

La începutul fiecărui capitol este prezentată o scurtă introducere, făcându-se o încadrare în contextul problemelor precedente precum și următoare.

Fiecare capitol se încheie cu teme propuse. Invităm cititorul să încerce să rezolve toate aceste probleme, scopul lor fiind de fixare și (auto-)evaluare a cunoștințelor dobândite.

Prezentăm în continuare o posibilă împărțire a lucrării în 14 unități de curs:

- Cursul 1.** Structura unui program C (capitolul 1)
- Cursul 2.** Tipuri numerice de date (capitolul 2)
- Cursul 3.** Funcții de scriere și citire în C (capitolul 3)
- Cursul 4.** Instrucțiuni de decizie, instrucțiuni repetitive, tipul char (capitolele 4, 5 și 6)
- Cursul 5.** Pointeri, tablouri de elemente (capitolul 7)
- Cursul 6.** Funcții în C (capitolul 8)
- Cursul 7.** String-uri (capitolul 9)
- Cursul 8.** Structuri și pointeri către structuri (capitolele 10 și 11)
- Cursul 9.** Uniuni și seturi de constante (capitolele 12 și 13)
- Cursul 10.** Fișiere în C (capitolul 14)
- Cursul 11.** Fluxuri standard în C și variabile statice (capitolele 15 și 16)
- Cursul 12.** Funcții cu listă variabilă de argumente (capitolul 17)
- Cursul 13.** Utilizarea modului text de afișare (capitolul 18)
- Cursul 14.** Grafică în C (capitolul 19)

CUPRINS

1. Structura unui program C	6
1.1. Incluseri	7
1.2. Constante. Macrocomenzi.	7
1.3. Asocieri de nume unor tipuri de date	8
1.4. Funcția main	9
2. Tipuri numerice de date	10
2.1. Tipuri întregi de date	10
2.2. Operatorii din C pentru valori întregi	11
2.3. Tipuri reale de date	12
2.4. Operatorii C pentru valori reale	12
2.5. Alți operatori în C	13
3. Funcții de scriere și citire în C	14
3.1. Funcția printf	14
3.2. Funcția scanf	16
4. Instrucțiuni de decizie	18
4.1. Instrucțiunea if	18
4.2. Instrucțiunea switch	19
5. Instrucțiuni repetitive	20
5.1. Instrucțiunea for	20
5.2. Instrucțiunea while	21
5.3. Instrucțiunea do ... while	22
6. Tipul char	23
7. Pointeri. Tablouri de elemente	24
7.1. Alocarea statică a memoriei	25
7.2. Alocarea dinamică a memoriei	26
8. Funcții în C	31
9. String-uri	36
10. Structuri	39
11. Pointeri către structuri. Liste înlanțuite	41
12. Uniuni	45
13. Seturi de constante (tipul enum)	46
14. Fișiere în C	47
14.1. Funcții de lucru cu fișiere	48
15. Fluxuri standard în C	58

16. Variabile statice	59
17. Funcții cu listă variabilă de argumente	60
18. Utilizarea modului text de afișare al adaptorului video	62
19. Grafică în C.....	65
19.1. Inițializarea modului grafic	65
19.2. Instrucțiuni pentru desenare de linii drepte	72
19.3. Instrucțiuni pentru desenare de curbe eliptice.....	74
19.4. Instrucțiuni pentru afișare de texte în modul grafic	74
19.5. Instrucțiuni pentru desenare de figuri umplute	76
19.6. Instrucțiuni pentru paleta de culori	82
19.7. Pagini grafice.....	83
19.8. Citire / scriere zonă de memorie video	87
19.9. Ferestre grafice	91
ANEXA 1 - Folosirea mouse-ului sub DOS	93
ANEXA 2 - Urmărirea execuției unui program. Rulare pas cu pas.	98
BIBLIOGRAFIE	100

Introducere

Limbajul C a fost lansat în anul 1972 în laboratoarele Bell de către Dennis Ritchie pentru sistemul de operare Unix. În anul 1973 limbajul C a ajuns suficient de puternic încât mare parte din sistemul Unix a fost rescris în C.

Limbajul C s-a extins rapid pe mai multe platforme și s-a bucurat încă de la început de un real succes datorită ușurinței cu care se puteau scrie programe.

La sfârșitul anilor 1980 a apărut limbajul C++ ca o extensie a limbajului C. C++ preia facilitățile oferite de limbajul C și aduce elemente noi, dintre care cel mai important este noțiunea de clasă, cu ajutorul căreia se poate scrie cod orientat pe obiecte în adevăratul sens al cuvântului.

În anul 1989 este finalizat standardul ANSI C. Standardul ANSI (American National Standards Institute) a fost adoptat în 1990 cu mici modificări și de către ISO (International Organization for Standardization).

Există multe compilatoare de C/C++. Dintre acestea, de departe cele mai cunoscute sunt compilatoarele – medii de programare integrate produse de firmele Borland și Microsoft.

Prima versiune a bine-cunoscutului mediu de programare Turbo C a fost lansată de către firma Borland în anul 1987. Standardul ANSI C / ISO C a constituit baza elaborării de către firma Borland a diferitelor sale versiuni de medii de programare, dintre care ultima a fost Borland C++ 5.02. În prezent firma Borland dezvoltă mediul de programare Borland C++ Builder și mediile Turbo C++ Professional și Explorer, dintre care ultimul este gratuit începând cu anul 2006.

În anul 1992 firma Microsoft a lansat prima versiune a mediului său de programare Visual C++, iar în anul 2002 a fost lansată prima versiune Visual C++ pentru platforma .NET.

În această carte vor fi prezentate limbajele C și C++ dintr-o perspectivă atât Borland, cât și Microsoft. Pe alocuri vor fi punctate micile diferențe dintre cele două compilatoare.

1. Structura unui program C

Obiective

Ne propunem să vedem cum se scrie un program în limbajul C, ce secțiuni apar, unde și cum se redactează.

Un program C se salvează de obicei cu extensia *C* și, implicit, compilarea se va face cu compilatorul C. Dacă, însă, programul este salvat cu extensia *CPP*, atunci compilarea se va face folosind compilatorul C++.

Un program C obișnuit are următoarea structură:

```
/* includeri
   definitii macrocomenzi
   definitii de tipuri de date
   declaratii de variabile globale
   definitii de constante globale
   definitii de functii sau/si descrierea unor functii */

void main()
{
    /* corpul functiei principale */
}

/* descriere functii care au definitia (antetul) deasupra functiei main */
```

Ordinea de apariție a includerilor, a definițiilor de macrocomenzilor, a tipurilor de date, a variabilelor și a constantelor globale este opțională. Mai mult, pot apărea mai multe grupuri de includeri, macrocomenzi, definiții de tipuri de date, variabile și constante. Astfel, de exemplu, putem avea declarații de variabile, apoi includeri, apoi iar declarații de variabile etc.

De regulă la începutul unui program C se pun includerile, dar, după cum am spus, nu este obligatoriu.

Să facem observația că între */** și **/* în C se pun comentariile (așa cum se poate vedea în programul de mai sus). Cu alte cuvinte, tot ce apare între aceste semne nu este luat în considerare la compilare.

Spre exemplificare prezentăm un program simplu care afișează un mesaj de salut:

```
# include <stdio.h> /* includerea fișierului antet stdio.h */

void main()
{
    printf("HELLO WORLD!");
    /* functia printf are antetul in fisierul stdio.h */
}
```

Facem observația că în C se face distincție între litere mari și mici, spre deosebire de limbajul Pascal, de exemplu, unde nu contează dacă redactăm codul cu litere mari sau cu litere mici. Astfel, în programul de mai sus, dacă scriam funcția *printf* cu litere mari, ea nu era recunoscută la compilare și obțineam un mesaj de eroare.

1.1. Incluseri

La începutul unui program C sau C++ se includ de obicei fișiere antet. Un fișier antet se recunoaște ușor prin faptul că are de obicei extensia *h*. Se pot include și fișiere cu extensia *C* sau *CPP*, care conțin cod C, respectiv cod C++.

Fișierele antet conțin în general numai definițiile unor funcții a căror implementare (descriere) se regăsește separat în fișiere cu extensiile *C*, *CPP*, *obj* sau *lib*. Fișierele **.obj* conțin cod C sau C++ în forma compilată cu compilatorul Borland C/C++. Fișierele **.lib* conțin biblioteci de funcții C și C++. Odată cu mediul de programare C/C++ se instalează și o mulțime de biblioteci de funcții împreună cu fișiere antet ce conțin definițiile acestor funcții.

Cele mai des incluse fișiere antet din C sunt *stdio.h* și *conio.h*, care conțin definițiile funcțiilor standard de intrare / ieșire (citiri, scrieri), respectiv definiții de funcții consolă I/O (de exemplu funcția *getch()*, care așteaptă apăsarea unui buton de la tastatură).

Includerea unui fișier începe cu semnul # (diez), urmat de cuvântul *include* și de numele fișierului care este dat între semnele < (mai mic) și > (mai mare), sau între ghilimele. Numele fișierului inclus poate fi precedat de eventuala cale unde se găsește el pe disc.

Dacă fișierul ce se include nu este precedat de cale și numele său este dat între ghilimele, atunci el este căutat în calea curentă sau în directorul cu fișierele antet ce se instalează odată cu mediul de programare.

Dacă fișierul inclus nu este precedat de cale și numele lui este dat între semnele < și >, atunci el este căutat numai în directorul cu fișierele antet ale mediului de programare C/C++.

Pentru limbajul C o parte dintre funcții sunt incluse implicit (compilatorul C le cunoaște fără a fi nevoie includerea vreunui fișier). Totuși, în funcție de compilator, pot fi generate mesaje de atenționare (warning) dacă nu facem includerile. În exemplul de mai sus dacă se salvează programul cu extensia *C* (și implicit se folosește compilatorul C), atunci includerea fișierului antet *stdio.h* nu este neapărat necesară. Dacă salvăm însă fișierul cu extensia *CPP*, atunci includerea lui *stdio.h* este obligatorie.

Iată în final și câteva exemple de includeri:

```
# include <stdio.h>
# include "conio.h"
# include "c:\bc\test\test.cpp"
```

1.2. Constante. Macrocomenzi.

În C o constantă obișnuită se poate defini folosind cuvântul rezervat *const* astfel:

```
const tip_date c=expresie_constanta;
```

Dacă lipsește tipul de date de la definirea unor constante, se consideră implicit tipul *int*.

Iată și câteva exemple:

```
const int x=1+4;
const a=1,b=2*2; /* tot constante întregi ! */
const double pi=3.14159
```

Macrocomanda reprezintă o generalizare a conceptului de constantă, în sensul că putem defini expresii constante, care se înlocuiesc în codul executabil în momentul compilării. Definirea unei macrocomenzi începe cu semnul # (diez) urmat de cuvântul *define*.

Dăm câteva exemple de macrocomenzi:

```
# define N 100
# define PI 3.14159
# define suma(x,y) x+y
# define alipire(a,b) (a##b)
```

Primele două macrocomenzi definesc constante obișnuite, N este o constantă de tip întreg, iar PI este una de tip real.

Ultimele două macrocomenzi de mai sus definesc câte o expresie constantă. Într-un program care cunoaște a treia macrocomandă, un apel de forma *suma(1.7, a)* se va înlocui la compilare cu $1.7+a$. Înlocuirea se face în fiecare loc în care este apelată macrocomanda. De aceea, macrocomanda poate fi considerată o generalizare a conceptului de constantă, deși apelul ei seamănă cu cel al unei funcții.

Comportamentul macrocomenzilor poate conduce la erori de programare pentru cei care nu cunosc modul lor de funcționare. De exemplu, valoarea expresiei *suma(1,2)*5* este 11 și nu 15, deoarece înlocuirea directă a apelului *suma(1,2)* la compilare cu $1+2$ conduce la expresia $1+2*5$, care are evident valoarea 11 și nu la expresia $(1+2)*5$ cu valoarea 15.

În exemplul de mai sus la ultima macrocomandă s-a folosit operatorul `##` care alipește (concatenează) doi tokeni. Astfel, un apel de forma *alipire(x,yz)* se înlocuiește cu *xyz*, care poate fi o variabilă, numele unei funcții etc.

Este important de observat că datorită comportamentului diferit de cel al funcțiilor, macrocomenzile sunt contraindicate pentru a fi folosite prea des într-un program, deoarece fiecare apel de macrocomandă se înlocuiește în memorie cu corpul macrocomenzii, ceea ce conduce la mărirea codului executabil. În concluzie, când se lucrează cu macrocomenzi codul C sau C++ se poate reduce ca lungime, dar codul în formă compilată poate crește.

Frumusețea macrocomenzilor constă în faptul că nu lucrează cu tipuri de date prestabilite. Astfel, în exemplul de mai sus macrocomanda *suma* va putea fi folosită pentru orice tip de date, atâta timp cât între x și y se poate aplica operatorul $+$. Din cauză că la definire nu se specifică tipul de date al parametrilor, macrocomenzile pot fi văzute ca un rudiment de programare generică oferit de limbajul C.

În limbajul C++ pot fi scrise funcții și clase șablon pentru care unul sau mai multe tipuri de date sunt nespecificate, identificarea acestor tipuri făcându-se în momentul compilării programului. Șabloanele reprezintă un suport pentru programare generică în adevăratul sens al cuvântului. Asupra șabloanelor vom reveni.

1.3. Asocieri de nume unor tipuri de date

În C putem asocia un nume unui tip de date cu ajutorul cuvântului rezervat *typedef* astfel:

```
typedef tip_de_date nume_asociat;
```

Dăm în continuare două exemple ilustrative:

```
typedef int intreg;
typedef struct nume_structura nume_structura;
```

Tipului numeric *int* i se asociază numele *intreg*. După această asignare, putem folosi cuvântul *intreg*, în loc de *int*.

În al doilea exemplu tipului de date de tip structură *struct nume_structura* (care trebuie să existe anterior definit) i se asociază denumirea *nume_structura* (fără *struct*). Acest lucru este des folosit în C pentru simplificarea scrierii tipului structură. Această asociere nu este necesară în

C++, unde tipul *struct nume_structura* poate fi folosit și fără cuvântul *struct*, fără o definire prealabilă cu *typedef*.

Asocierile de nume unor tipuri de date se pot face atât în exteriorul, cât și în interiorul funcțiilor.

1.4. Funcția main

În orice program C/C++ trebuie să existe o unică funcție *main* (principală), din interiorul căreia începe execuția aplicației.

Pentru a înțelege mai bine modul de definire al funcției *main* vom prezenta pe scurt câteva generalități legate de modul de definire al unei funcții oarecare.

În C și C++ nu există proceduri, ci numai funcții. În definiția unei funcții tipul returnat se pune înaintea numelui funcției. Dacă în momentul definirii funcției se omite tipul returnat, nu este greșit și se consideră implicit tipul returnat ca fiind *int*. O funcție care are ca tip returnat *void* (vid) se apelează ca o procedură. Dacă o funcție nu are parametri, după nume se pun paranteze rotunde sau se pune cuvântul rezervat *void* între paranteze rotunde în momentul definirii ei.

Funcția *main* poate avea ca tip returnat *void* sau *int* (care se pune înaintea cuvântului *main*). Dacă tipul returnat este *int*, atunci în interiorul funcției *main* pot apărea instrucțiuni de forma *return val_int* care au ca efect întreruperea execuției programului și returnarea către sistemul de operare a valorii întregi *val_int*. În această situație, ultima instrucțiune din funcția *main* este de obicei *return 0*, ceea ce înseamnă că programul s-a terminat cu succes. Dacă apare vreo eroare pe parcursul execuției programului care nu poate fi tratată (memorie insuficientă, imposibilitate de deschide a unui fișier etc.), programul se părăsește în general cu o instrucțiune de forma *return val_int*, unde *val_int* este o valoare întreagă nenulă. Astfel, semnalăm sistemului de operare faptul că execuția programului s-a încheiat cu insucces (cu eroare).

Funcția *main* poate să nu aibă nici un parametru, sau poate avea 2 parametri. Dacă funcția *main* are doi parametri, atunci primul este de tip *int* și reține numărul de parametri în linie de comandă cu care s-a executat programul, iar al doilea este un șir de string-uri, în care sunt memorați parametrii de apel în linie de comandă. În primul *string* (pe poziția 0 în vector) se reține numele executabilului (al aplicației), după care urmează parametrii de apel în linie de comandă.

Presupunem că avem programul C *test* care are următoarea funcție *main*:

```
void main(int nargs, char argv *arg[])
{
    /* .... */
}
```

Apelăm programul *test* în linie de comandă cu doi parametri:

```
test param1 param2
```

În această situație, în funcția *main* a programului *test* vom avea:

- *nargs* va avea valoarea 3
- *arg[0]* va fi "*test.exe*"
- *arg[1]* va fi "*param1*"
- *arg[2]* va fi "*param2*".

Asupra modului de definire și descriere a funcțiilor o să revenim.

Rezumat

Înainte de funcția *main* pot apărea niciuna, una sau mai multe secțiuni de tipul:

- **incluseri.** Pot fi incluse în general fișiere antet (cu extensia **.h**), dar pot apărea și fișiere cu extensia **.c** sau **.cpp**. Aceste fișiere conțin în general antete (definiții) de funcții, definiții de tipuri de date și constante, macrocomenzi, dar pot apărea și implementări de funcții, variabile globale etc.
- **definiții de macrocomenzi.** Macrocomanda reprezintă o generalizare a conceptului de constantă. O macrocomandă este o expresie constantă. Fiecare apel de macrocomandă este înlocuit la compilare cu corpul macrocomenzii. O macrocomandă se descrie după **#define**.
- **definiții de tipuri de date.** Unui tip nou de date definit de programator i se poate da un nume folosind declarația care începe cu cuvântul rezervat **typedef**.
- **declarații de variabile.**
- **definiții de constante globale.** Constantele în C se definesc după cuvântul rezervat **const**. O constantă obișnuită poate fi definită și ca o macrocomandă.
- **definiții sau/și descrieri de funcții.** Funcțiile programului C se declară deasupra funcției *main* și se implementează după funcția *main*, sau se descriu în întregime deasupra funcției *main*.

Funcția **main** poate avea ca tip returnat tipul **void** sau tipul **int**. Funcția **main** poate să nu aibă nici argumente sau poate avea două argumente de apel, argumente care memorează parametrii de apel în linie de comandă ai aplicației.

Teme

1. Folosind operatorul **?:** (vezi subcapitolul 2.5) scrieți o macrocomandă pentru maximum dintre două numere.
2. Scrieți o macrocomandă pentru maximum dintre trei valori.
3. Folosind prima macrocomandă scrieți o macrocomandă pentru maximum dintre patru valori.
4. Ce trebuie să conțină parametrii **narg** și **argv** pentru o aplicație care ar calcula media aritmetică în linie de comandă a oricâtor valori reale ?

2. Tipuri numerice de date

Obiective

Ne propunem să facem cunoștință cu tipurile numerice de date ale limbajului C, modul lor de reprezentare în memorie, domeniul de valori, operatorii specifici acestor tipuri de date și alți operatori pe care îi întâlnim în C.

În C și C++ avem două categorii de tipuri numerice de date: tipuri întregi și reale. Cu cele două clase de tipuri numerice se lucrează diferit (la nivel de procesor). Reprezentarea informației în memorie este diferită, avem operatori diferiți.

2.1. Tipuri întregi de date

În tabelul de mai jos sunt prezentate tipurile numerice întregi cu semn (signed) și fără semn (unsigned) din C. Facem observația că numărul de octeți pe care se reprezintă în memorie valorile întregi și implicit domeniile de valori din tabelul următor sunt cele pentru Windows, mai exact, cele din Visual C:

Denumire tip	Număr octeți	Domeniu de valori
(signed) char	1	-128 la 128
unsigned char	1	0 la 255
enum	2	-32.768 la 32.767
short (signed) (int)	2	0 la 65.535
short unsigned (int)	2	-32.768 la 32.767
(signed) int	4	-2.147.483.648 la 2.147.483.647
unsigned int	4	0 la 4.294.967.295
(signed) long	4	-2.147.483.648 la 2.147.483.647
unsigned long	4	0 la 4.294.967.295

Să facem câteva observații:

- În Visual C tipul *enum* coincide cu tipul *short int*, iar tipul *int* coincide cu *long*.
- Tot ceea ce apare între paranteze rotunde în tabelul de mai sus este opțional.
- Tipurile numerice de date au o dublă utilizare în C și C++: pentru valori numerice (numere) și pentru valori booleene. Astfel, o valoare numerică nenulă (întreagă sau reală) corespunde în C/C++ valorii booleene de adevărat (*true*), iar o valoare numerică nulă corespunde valorii booleene de fals (*false*).
- Tipurile *char* și *unsigned char* în C și C++ au o triplă întrebuințare: pentru valori întregi pe un octet, pentru valori booleene și pentru caractere. Valoarea întreagă de tip *char* reprezintă codul ASCII al unui caracter. Astfel, constanta de tip *char* 'a' reprezintă valoarea întreagă 97, care este de fapt codul ASCII al caracterului *a*. Cu alte cuvinte, în C 'a' este același lucru cu 97 !
- În Visual C pentru caractere Unicode este definit tipul *wchar_t* pe doi octeți.

2.2. Operatorii din C pentru valori întregi

Operatorul = (egal) este folosit pentru atribuirea valorii unei expresii întregi unei variabile întregi:

```
i = expresie;
```

Operatorul returnează valoarea atribuită variabilei *i*, valoare rezultată în urma evaluării expresiei și conversiei la tipul variabilei *i*, în cazul nostru fiind vorba de tipul *int*. Astfel, de exemplu *i = 1 + sqrt(2)* este o expresie care are valoarea 2.

Pentru tipurile întregi de date sunt definiți **operatorii aritmetici binari**: + (adunare), - (scădere), * (înmulțire), / (câtul împărțirii), % (restul împărțirii), +=, -=, *=, /=, %=, unde expresia *x += y* este echivalentă cu *x = x+y*.

Operatori aritmetici unari definiți pentru tipurile întregi de date sunt: ++ și --. Ei realizează incrementarea, respectiv decrementarea unei variabile întregi. Astfel, *x++* este echivalent cu *x = x+1*, iar *x--* cu *x = x-1*. Cei doi operatori unari se folosesc sub două forme: *x++*, *x--*, adică forma postincrementare, respectiv postdecrementare, iar *++x*, *--x* sunt sub forma

preincrementare, respectiv predecrementare. Să explicăm acum care este diferența dintre cele două forme de utilizare a operatorului de incrementare.

Presupunem că variabila y reține valoarea 1. Pentru o expresie de forma $x = ++y$, întâi se incrementează y și apoi noua valoare a lui y se atribuie lui x și, în consecință, x devine 2. Dacă operatorul $++$ se folosește în forma postincrementată, adică $x = y++$, atunci lui x întâi i se atribuie valoarea variabilei y și abia după aceea se face incrementarea lui y . În această situație variabila x va fi inițializată cu valoarea 1. Operatorul de decrementare $--$ funcționează în mod similar ca și $++$, tot sub două forme, predecrementare și postdecrementare.

Operatorii relaționali sunt: $<$ (mai mic), $<=$ (mai mic sau egal), $>$ (mai mare), $>=$ (mai mare sau egal), $==$ (egalitate), $!=$ (diferit). Cu alte cuvinte, cu ajutorul acestor operatori, două valori întregi se pot compara.

Operatorul ! (semnul exclamării) aplicat unei valori numerice are efectul *not*, adică din punct de vedere boolean îi schimbă valoarea (din False în True, respectiv din True în False). Astfel, $!x$ are valoare 0 dacă și numai dacă x este nenul.

Operatori la nivel de bit aplicabili numai pe valori întregi sunt: $\&$ (și pe biți), $|$ (sau pe biți), \wedge (sau exclusiv), $>>$ (*shift*-are biți la dreapta), $<<$ (*shift*-are biți la stânga), \sim (not pe biți). Pentru a înțelege cum funcționează acești operatori dăm următorul exemplu:

Fie a și b două variabile întregi pe un octet, a reprezentat pe biți este (1,0,1,1,0,1,1,1) și $b = (0,1,1,1,0,0,1,0)$.

Avem: $a \& b = (0,0,1,1,0,0,1,0)$, $a | b = (1,1,1,1,0,1,1,1)$, $a \wedge b = (1,1,0,0,0,1,0,1)$, $a >> 1 = (0,1,0,1,1,0,1,1)$, $a << 1 = (0,1,1,0,1,1,1,0)$, $\sim a = (0,1,0,0,1,0,0,0)$.

Facem observația că dacă a este întreg cu semn (de tip *char*), atunci $a >> 1 = (1,1,0,1,1,0,1,1)$, deoarece cel mai semnificativ bit (cel mai stânga) reprezintă semnul numărului întreg și în consecință după *shift*-are la dreapta rămâne tot 1 (corespunzător semnului minus). Să observăm că *shift*-area cu n biți a unei valori întregi este echivalentă cu o împărțire la 2^n . În cazul exemplului nostru $a >> 1$ este echivalent cu $a = a / 2$.

Pentru operatorii pe biți avem și variantele combinate cu atribuire: $\&=$, $|=$, $\wedge=$, $<<=$, $>>=$, unde $x \&= y$ este echivalent cu $x = x \& y$ etc.

2.3. Tipuri reale de date

În tabelul de mai jos sunt trecute tipurile numerice reale (în virgulă mobilă) din C (denumirea, numărul de octeți pe care se reprezintă în memorie și domeniul de valori în valoare absolută):

Denumire tip	Număr octeți	Domeniu de valori în valoare absolută
float	6	$3,4 \cdot 10^{-38}$ la $3,4 \cdot 10^{38}$
double	8	$1,7 \cdot 10^{-308}$ la $1,7 \cdot 10^{308}$
long double	10	$3,4 \cdot 10^{-4932}$ la $1,1 \cdot 10^{4932}$

Desigur, este importantă și precizia cu care lucrează fiecare din tipurile numerice reale de mai sus (numărul maxim de cifre exacte). Precizia cea mai bună o are tipul *long double* și cu cea mai scăzută precizie este tipul *float*.

2.4. Operatorii C pentru valori reale

Pentru tipurile reale de date în C sunt definiți operatorii:

- **Operatorul de atribuire** =.
- **Operatorii aritmetici binari**: + (adunare), - (scădere), * (înmulțire), / (împărțire).
- **Operatorii aritmetici binari combinați cu atribuire** +=, -=, *=, /=.
- **Operatorii relaționali**, **not** și de **incrementare**, respectiv **decrementare** funcționează și pentru valori reale.

Operatorii pe biți nu funcționează pentru valori reale!

2.5. Alți operatori în C

Operatorii logici: && (și), || (sau), ! (not). Dăm un exemplu de expresie logică: (a && !(b%2) && (a>b || b<=0)).

Operatorul ?: este singurul operator ternar (funcționează cu trei operanzi) din C și se folosește astfel: *var* = (*expresie_logică*) ? *val1:val2* cu semnificația: se evaluează *expresie_logică*, dacă expresia are valoare de *adevărat* (diferită de 0), atunci variabila *var* ia valoarea *val1*, altfel *var* primește valoarea *val2*.

Operatorul [] este pentru acces la elementul unui vector. De exemplu, *a[1]* reprezintă al doilea element al șirului *a*, deoarece în C indicii vectorilor încep cu 0. Este interesant faptul că în C dacă scriem *l[a]* înseamnă același lucru !

Operatorul () este folosit pentru a realiza o conversie de tip (exemplu: *(float)n*) convertește valoarea din variabila *n* la tipul *float*, dacă acest lucru este posibil).

Operatorii . (punct) și **->** sunt pentru acces la membrul unei structuri, uniuni sau clase (al doilea pentru cazul în care se lucrează cu pointeri).

Operatorul sizeof dă dimensiunea în octeți pe care o ocupă în memorie valoarea rezultată în urma evaluării unei expresii. De exemplu, expresia *sizeof 1+2.5* are valoarea 6, deoarece rezultatul expresiei *1+2.5* este o valoare de tip *float*, care se reprezintă în memorie pe 6 octeți. În C există și funcția *sizeof(tip)* care returnează numărul de octeți necesari pentru stocarea în memorie a unei valori de tipul *tip*. De exemplu, *sizeof(short)* returnează valoarea 2.

```
float x;  
char s[100];  
printf("%d == %d\n",sizeof x,sizeof(float));  
printf("%d != %d\n",sizeof s,sizeof(char*));
```

În exemplul de mai sus *sizeof x* și *sizeof(float)* au aceeași valoare, adică 6, deoarece amândouă reprezintă numărul de octeți pe care este reprezentat în memorie un număr real de tip *float*. În schimb, cu toate că *char** este tipul variabilei *s*, *sizeof s* returnează numărul de octeți rezervați pentru șirul *s*, adică 100, iar *sizeof(char*)* returnează valoarea 4, adică numărul de octeți pe care se reprezintă în memorie o adresă (un pointer) către tipul *char*.

Operatorul # (diez) este folosit pentru directive preprocesor (a se vedea discuția referitoare la macrocomenzi).

Operatorul ## realizează alipirea a doi tokeni (a fost menționat tot la discuția legată de macrocomenzi).

Operatorul , (virgulă) se aplică între două expresii care se evaluează după regula: se evaluează întâi prima expresie și apoi a doua (cea din dreapta), rezultatul returnat de operator fiind valoarea ultimei expresii. De exemplu, expresia *x = 2, x - 4* are valoarea -2. De obicei, o expresie în care apare operatorul virgulă se pune între paranteze pentru eliminarea eventualelor ambiguități. Dacă folosim de mai multe ori operatorul virgulă: *expr1, expr2, ... , exprn*, atunci

execuția se face de la stânga spre dreapta. De exemplu, expresia $x=1$, $x+4$, $x*2$ are valoarea 2, iar expresia $x=1$, $x+=4$, $x*2$ are valoarea 10.

Rezumat

Tipurile întregi de date ale limbajului C sunt: **(signed) char**, **unsigned char**, **enum**, **short (signed) (int)**, **short unsigned (int)**, **(signed) int**, **unsigned int**, **(signed) long** și **unsigned long**. De departe cel mai cunoscut și utilizat tip de date din C este *int*, care se reprezintă pe doi sau patru octeți, în funcție de compilator și de sistemul de operare. Operatorii aritmetici binari pentru valori întregi sunt: + (adunare), - (scădere), * (înmulțire), / (câtul împărțirii) și % (restul împărțirii).

Tipurile întregi de date pot fi folosite și pentru a reține caractere. Practic, valoarea numerică întreagă în C se identifică cu caracterul având codul ASCII acea valoare numerică.

Tipurile reale de date din C sunt: **float**, **double** și **long double**. Operatorii aritmetici binari pentru valori reale sunt: + (adunare), - (scădere), * (înmulțire) și / (împărțire).

Tipurile numerice de date (întregi și reale) în C sunt și tipuri booleane. Orice valoare numerică diferită de zero corespunde valorii booleane de adevărat, iar orice valoare numerică nulă (zero) corespunde valorii booleane de fals.

Teme

1. Folosind operatorul **sizeof** calculați media aritmetică a numărului de octeți pe care se reprezintă cele nouă tipuri întregi de date.
2. Aceeași problemă pentru cele trei tipuri reale de date.

3. Funcții de scriere și citire în C

Obiective

Ne propunem să vedem cum putem în C afișa mesaje pe ecran și cum putem prelua date de la tastatură pentru variabile.

Definițiile funcțiilor pentru scrierea și citirea datelor în limbajului C le găsim în fișierul antet *stdio.h*.

3.1. Funcția printf

Funcția *printf* este folosită pentru afișarea formatată a unui mesaj pe ecranul monitorului. Ea are următoare structură:

```
printf(constanta_sir_de_caractere, lista_expresii);
```

Funcția *printf* afișează pe ecranul monitorului *constanta_sir_de_caractere* în care toți descriptorii de format se înlocuiesc cu valorile expresiilor, înlocuirea făcându-se de la stânga spre dreapta.

Dăm un exemplu:

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

```

void main()
{
    int n=10;
    float x=5.71;
    printf("Un nr. intreg: %d si un nr. real: %f\n",n,x);
}

```

În C, în momentul declarării variabilelor, ele pot fi și inițializate (așa cum am procedat în exemplul de mai sus pentru n și x).

$\%d$ și $\%f$ sunt specificatori (descriptori) de format pentru tipurile de date *int* și respectiv *float*. Descriptorii specifică faptul că în locul în care se află în șirul de caractere ce se afișează pe ecran vor apărea valori de tipul indicat, valori ce vor fi luate din lista de expresii aflată după șirul de caractere. În exemplul nostru lista de expresii este alcătuită din valorile n și x .

Înlocuirea descriptorilor în constanta de tip șir de caractere (primul parametru al funcției *printf*) se face de la stânga spre dreapta. În exemplul de mai sus $\%d$ va fi înlocuit cu valoarea reținută în variabila n , adică 10 , iar $\%f$ se înlocuiește cu valoarea lui x , adică 5.71 .

Dacă tipurile expresiilor ce se înlocuiesc nu corespund cu descriptorii de format, sau dacă numărul descriptorilor nu este egal cu cel al expresiilor ce se înlocuiesc, nu se semnalează eroare, nici măcar atenționare la compilare, dar afișarea va fi eronată!

Dăm în continuare lista cu descriptorii de format sau descriptorii de tip, cum se mai numesc:

$\%c$	– caracter (<i>char</i>)
$\%s$	– șir de caractere încheiat cu caracterul ‘\0’ (string)
$\%d$	– întreg cu semn (<i>int</i>)
$\%u$	– întreg fără semn (<i>unsigned int</i>)
$\%ld$	– întreg lung (pe 4 octeți) cu semn (<i>long</i>)
$\%lu$	– întreg lung (pe 4 octeți) fără semn (<i>unsigned long</i>)
$\%x$	– întreg în baza 16 fără semn (<i>int</i>) (cifrele în hexazecimal pentru $10, 11, 12, 13, 14, 15$ sunt litere mici, adică a, b, c, d, e și f)
$\%X$	– la fel ca $\%x$, numai ca cifrele în hexazecimal pentru sunt litere mari
$\%o$	– întreg în baza 8 fără semn (<i>int</i>)
$\%f$	– real pe 6 octeți (<i>float</i>), notație zecimală (fără exponent)
$\%e$	– real pe 6 octeți (<i>float</i>), notație exponențială, științifică (litera e de la exponent este mică)
$\%E$	– la fel ca $\%e$, numai că pentru litera de la exponent este mare
$\%g$	– real pe 6 octeți (<i>float</i>), notație zecimală sau exponențială, care este mai scurtă, iar dacă se afișează exponențial, atunci litera de la exponent este e
$\%G$	– real pe 6 octeți (<i>float</i>), notație zecimală sau exponențială, care este mai scurtă, iar dacă se afișează exponențial, atunci litera de la exponent este E
$\%lf$	– real pe 8 octeți (<i>double</i>), notație zecimală
$\%le$	– real pe 8 octeți (<i>double</i>), notație exponențială (litera exponent e este mică)
$\%lE$	– la fel ca la $\%le$, numai litera exponent E este mare
$\%lg$	– real pe 8 octeți (<i>double</i>), notație zecimală sau exponențială, care e mai scurtă, dacă e cazul se folosește literă mică pentru exponent
$\%lG$	– real pe 8 octeți (<i>double</i>), notație zecimală sau exponențială, care e mai scurtă, dacă e cazul se folosește literă mare pentru exponent
$\%Lf$	– real pe 10 octeți (<i>long double</i>), notație zecimală
$\%Le$	– real pe 10 octeți (<i>long double</i>), notație exponențială (litera de la exponent este mică, adică e)
$\%LE$	– real pe 10 octeți (<i>long double</i>), notație exponențială (litera de la exponent este mare, adică E)

- %Lg – real pe 10 octeți (*long double*), notație zecimală sau exponențială, care este mai scurtă, literă mică pentru exponent
- %LG – real pe 10 octeți (*long double*), notație zecimală sau exponențială, care este mai scurtă, literă mare pentru exponent
- %p – adresa în hexazecimal (pentru pointeri).

La afișarea folosind funcția *printf* se pot face formatări suplimentare, pornind de la un descriptor elementar. Formatările se referă la numărul de caractere pe care se face afișarea unei valori, tipul alinierii, caracterele ce se completează în locurile libere (spații, zerouri) etc. Dăm în acest sens câteva exemple:

%50s realizează afișarea unui string pe 50 caractere cu aliniere la dreapta, iar %-50s face același lucru, dar cu aliniere la stânga. De obicei string-urile se aliniază la stânga.

%4d realizează afișarea unui număr întreg pe 4 caractere cu aliniere la dreapta, iar %-4d face același lucru, dar cu aliniere la stânga. De obicei valorile numerice se aliniază la dreapta.

%10.2f realizează afișarea unui număr real pe 10 caractere cu 2 cifre exacte după virgulă, cu aliniere la dreapta, iar %-10.2f face același lucru, dar cu aliniere la stânga.

%010.2f realizează afișarea unui număr real pe 10 caractere cu 2 cifre exacte după virgulă, cu aliniere la dreapta, iar spațiile goale (din fața numărului) se completează cu zerouri. De exemplu, numărul 1.4 se va afișa sub forma 0000001.40. Dacă nu se pune 0 în fața lui 10.2f, atunci în loc de zerouri se completează implicit spații.

În programul de mai sus (afișarea unui întreg și a unui număr real), după afișarea pe ecran a mesajului, se sare la începutul liniei următoare, deoarece șirul s-a încheiat cu \n. Combinația de caractere \n este una dintre așa numitele *secvențe escape*.

Lista secvențelor escape recunoscute de limbajul C este:

\n – salt la linie nouă (nu neapărat la începutul lui)

\r – salt la început de rând

\t – deplasare la dreapta (*tab*)

\b – deplasare la stânga cu un caracter (*backspace*)

\f – trecere pe linia următoare (*formfeed*)

\' – apostrof

\" – ghilimele

\\ – backslash

\xcc – afișarea unui caracter având codul ASCII în hexazecimal dat de cele două cifre de după \x (fiecare *c* reprezintă o cifră în baza 16). De exemplu, \x4A este caracterul cu codul ASCII 4A₍₁₆₎ = 74₍₁₀₎, adică litera J.

Este bine de reținut faptul că într-un șir constant de caractere, semnul \ (*backslash*) trebuie dublat. Asta se întâmplă mai ales când este vorba de căi de fișiere. De asemenea, semnele " (*ghilimele*) și ' (*apostrof*) trebuiesc precedate de semnul \ (*backslash*).

În C pentru a trimite un mesaj în fluxul standard de erori *stderr* folosim funcția **perror**, în loc de *printf*. Mesajul de eroare ajunge tot pe ecranul monitorului, dar pe altă cale. Datorită faptului că funcția *perror* nu suportă formatare așa cum face *printf*, în exemplele pe care o să le dăm în această carte vom folosi funcția *printf* pentru afișarea mesajelor de eroare.

3.2. Funcția scanf

Funcția *scanf* este folosită pentru preluarea formatată de la tastatură a valorilor pentru variabile. Funcția are următoarea structură:


```
scanf(const_sir_caractere, lista_adrese_var_citite);
```

constanta_sir_caractere este un string care conține numai descriptorii de tip ai variabilelor ce se citesc. Descriptorii sunt cei elementari pe care i-am prezentat mai sus (vezi funcția *printf*), adică: %c, %s, %d, %u, %ld, %lu, %x, %X, %o, %f, %e, %E, %g, %G, %lf, %le, %lE, %lg, %lg, %Lf, %Le, %LE, %Lg, %LG, %p.

În exemplul următor apar câteva citiri de la tastatură:

```
# include <stdio.h>

void main()
{
    int n;
    float f;
    char s[10]; /* sir de caractere */
    scanf("%d%f%s", &n, &f, s);
    printf("Am citit: %d, %f, %s", n, f, s);
}
```

Facem observația că prin *&var* se înțelege adresa la care se află reținută în memorie variabila *var*. În C valorile se returnează prin adresă prin intermediul parametrilor unei funcții.

Variabila *s* este deja un pointer către zona de memorie în care se memorează un șir de caractere, așadar nu este necesar să punem semnul & (și) în fața lui *s* la citire. Dacă se pune totuși semnul & în fața variabilei *s* la citire, nu se semnalează eroare nici măcar atenționare, “pleonasmul” în C în general sunt ignorate de compilator. Vom reveni cu mai multe explicații când vom vorbi despre pointeri și tablouri (șiruri).

Pentru funcțiile *printf* și *scanf* există și variantele pentru scriere, respectiv citire în/dintr-un string sau fișier text.

Numele funcțiilor pentru string-uri încep cu litera *s* (*sprintf*, respectiv *sscanf*) și funcționează absolut la fel ca cele obișnuite numai că destinația scrierii, respectiv sursa citirii este un string. În consecință mai apare un parametru suplimentar (variabila șir de caractere) în fața celor pe care îi au funcțiile *printf* și *scanf*.

Numele funcțiilor de scriere și citire pentru fișiere text încep cu litera *f* (*fprintf*, respectiv *fscanf*). Aceste funcții au un parametru suplimentar (variabila pointer către tipul *FILE*).

Asupra acestor funcții o să revenim atunci când o să discutăm despre string-uri, respectiv fișiere.

În Borland C pentru DOS există variantele *cprintf*, respectiv *cscanf* ale funcțiilor *printf*, respectiv *scanf* pentru scriere și citire formatată într-o fereastră text (creată cu funcția *window*). Facem observația că schimbarea culorii textului și a fundalului pe care se afișează un text are efect în cazul utilizării funcției *cprintf*, dar nu are efect atunci când se folosește funcția *printf*. De asemenea, pentru salt la începutul unei linii trebuie să punem pe lângă \n și secvența escape \r, ca să se revină la începutul rândului, ceea ce nu era necesar în cazul funcției *printf*, secvența escape \n fiind suficientă.

Trebuie precizat faptul că funcția *printf* nu ține cont de marginile ferestrei text definite cu *window*. Ca parametri, funcția *window* primește cele 4 coordonate ecran text ale colțurilor stânga-sus și respectiv dreapta-jos ale ferestrei.

Rezumat

Afișarea formatată a mesajelor pe ecran se face cu ajutorul funcției **printf**, iar preluarea formatată a datelor de la tastatură se face folosind funcția **scanf**. Pentru a utiliza aceste funcții trebuie să includem fișierul antet **stdio.h**.

Teme

1. Afișați pe ecranul monitorului cele două tabele cu tipurile numerice de date de la capitolul anterior.
2. De la tastatură citiți datele unor persoane (nume și prenume, vârstă și salariu). Afișați tabelar aceste date pe ecran după modelul următor:

Nr. crt.	NUMELE SI PRENUMELE	Varsta	Salariu
1	Ionescu_Monica	19	520.17
2	Aionesei_Adrian_Ionel	23	884.25
3	Popescu_Gigel	19	443.10
4	Popescu_Maria	28	1155.00
Medie		22.25	750.63

4. Instrucțiuni de decizie

Obiective

Ne propunem să vedem cum se redactează o instrucțiune de decizie în C.

În C există două instrucțiuni de decizie: *if ... else* și instrucțiunea de decizie multiplă (cu oricâte ramuri) *switch ... case*.

4.1. Instrucțiunea if

În C o instrucțiune de tipul *dacă ...atunci ...altfel* are următoarea structură:

```
if (conditie)
{
    /* grupul de instructiuni 1 */
}
else
{
    /* grupul de instructiuni 2 */
}
```

Semnificația instrucțiunii este: dacă este adevărată condiția, atunci se execută *grupul de instrucțiuni 1*, altfel se execută *grupul de instrucțiuni 2*. Ramura *else* poate lipsi.

Un grup de instrucțiuni în C se delimitează cu ajutorul acoladelor {}. Dacă un grup este format dintr-o singură instrucțiune, atunci acoladele pot lipsi.

Dăm un exemplu ilustrativ pentru instrucțiunea *if*:

```
float x=2.2, y=4;
int a=25, b=85;

if (x>y) max=x; else max=y;
if (a)
{
    printf("Restul impartirii este: %d\n",b%a);
    printf("Catul impartirii este:  %d",b/a);
}
else perror("Impartire prin 0!");
```

4.2. Instrucțiunea switch

switch este o instrucțiune decizională multiplă (cu mai multe ramuri). Structura ei este următoarea:

```
switch (expresie_intreaga)
{
    case val1:
        /* grupul de instructiuni 1 */
        break;
    case val2:
        /* grupul de instructiuni 2 */
        break;
    /* .... */
    case valn:
        /* grupul de instructiuni n */
        break;
    default:
        /* grupul de instructiuni n+1 */
}
```

Semnificația instrucțiunii este: Se evaluează expresia *expresie_intreaga* care trebuie să aibă o valoare întreagă. Avem următoarele situații:

- Dacă valoarea expresiei este egală cu *val1*, atunci se execută *grupul de instrucțiuni 1*.
- Dacă valoarea expresiei este egală cu *val2*, atunci se execută *grupul de instrucțiuni 2*.
-
- Dacă valoarea expresiei este egală cu *valn*, atunci se execută *grupul de instrucțiuni n*.
- Dacă valoarea obținută în urma evaluării expresiei nu este egală cu nici una dintre valorile *val1*, ... , *valn*, atunci se execută *grupul de instrucțiuni n+1*.

Să facem câteva observații:

- 1) La instrucțiunea *switch* grupurile de instrucțiuni de pe ramuri nu trebuie să fie delimitate cu acolade. Nu este greșit însă dacă le delimităm totuși cu acolade.
- 2) După fiecare grup de instrucțiuni punem în general instrucțiunea *break*. În lipsa instrucțiunii *break* se execută și instrucțiunile de pe ramurile de mai jos până la sfârșitul instrucțiunii *switch* (inclusiv cele de pe ramura *default*) sau până se

- întâlnește primul *break*. Instrucțiunea *break* întrerupe execuția instrucțiunii *switch* și a celor repetitive (*for*, *while* și *do ... while*).
- 3) Ramura *default* poate lipsi.
 - 4) Dacă există ramura *default*, nu este obligatoriu să fie ultima.
 - 5) Valorile *val1*, *val2*, ..., *valn* trebuie să fie constante întregi și distincte două câte două.

Spre exemplificare considerăm o secvență de program pentru calcularea într-un punct a valorii unei funcții f cu trei ramuri:

$$f: Z \rightarrow R, f(x) = \begin{cases} -1, & \text{dacă } x = 0 \\ -2, & \text{dacă } x = 1 \\ x+1, & \text{dacă } x \neq 1 \text{ și } x \neq 2 \end{cases}.$$

```
int x;
printf("Dati un numar întreg: ");
scanf("%d", &n);
printf("Valoarea functiei este f(%d)=", x);
switch (x)
{
    case 0:
        printf("-1"); break;
    case 1:
        printf("-2"); break;
    default:
        printf("%d", x+1); break;
}
```

Rezumat

În C avem două instrucțiuni de decizie: instrucțiunea **if**, care are una sau două ramuri (ramura **else** poate lipsi) și instrucțiunea **switch** (cu oricâte ramuri). Expresia după care se face selecția ramurii **case** trebuie să aibă o valoare întreagă. De asemenea, de reținut este și faptul că în principiu, fiecare ramură a instrucțiunii *switch* se termină cu un apel **break**.

5. Instrucțiuni repetitive

Obiective

Ne propunem să facem cunoștință cu instrucțiunile repetitive ale limbajului C.

În C există trei instrucțiuni repetitive: *for*, *while* și *do ...while*.

5.1. Instrucțiunea for

În C, instrucțiunea repetitivă *for* este destul complexă. Ea are următoarea structură:

```
for (expresie_1; expresie_2; expresie_3)
{
    /* grup de instructiuni */
}
```

Semnificația instrucțiunii este:

- 1) Se evaluează *expresie 1*. Această expresie conține în general inițializări de variabile.
- 2) Cât timp valoarea *expresiei 2* este adevărată (nenulă), se execută *grupul de instrucțiuni*. Această expresie reprezintă condiția de oprire a ciclării.
- 3) După fiecare iterație se evaluează *expresie 3*. Această expresie conține în general actualizări ale unor variabile (incrementări, decrementări etc.).

Prezentăm câteva caracteristici ale instrucțiunii *for*:

- 1) Oricare dintre cele 3 expresii poate lipsi. Lipsa *expresiei 2* este echivalentă cu valoarea 1 (de adevăr). Părăsirea ciclului *for* în această situație se poate face în general cu *break*.
- 2) În prima expresie pot apărea mai multe inițializări, care se dau cu ajutorul operatorului virgulă (vezi subcapitolul dedicat operatorilor). De asemenea, ultima expresie poate conține mai multe actualizări despărțite prin virgulă.

Ca aplicație pentru instrucțiunea *for*, calculăm suma primelor n numere întregi pozitive (valoarea lui n este citită de la tastatură) și cel mai mare divizor comun pentru două numere întregi (preluate tot de la tastatură) folosind algoritmul lui Euclid:

```
long s,i,n,a,b,x,y;

printf("n="); scanf("%ld",&n);
for (s=0,i=0; i<n; i++) s+=i; /* Calculare: 1+2+...+n */
printf("Suma primelor %ld numere intregi pozitive este %ld\n",n,s);

printf("Dati doua numere intregi: ");
scanf("%ld%ld",&a,&b);
for (x=a,y=b; y; r=x%y,x=y,y=r); /* Calculare Cmmdc(a,b) */
printf("Cmmdc(%ld,%ld)=%ld\n",a,b,x);
printf("Cmmmcc(%ld,%ld)=%ld\n",a,b,a/x*b);
```

5.2. Instrucțiunea while

Instrucțiunea *while* are următoarea structură:

```
while (conditie)
{
    /* grup de instructiuni */
}
```

Semnificația instrucțiunii este: cât timp *condiția* este adevărată, se execută *grupul de instrucțiuni*. Părăsirea ciclării se poate face forțat cu *break*.

Calcularea celui mai mare divizor comun a două numere întregi folosind instrucțiunea *while* poate fi scrisă astfel:

```

long a,b,x,y,r;
printf("Dati doua numere întregi: ");
scanf("%ld%ld",&a,&b);
x=a; y=b;
while (y) { r=x%y; x=y; y=r; }
printf("Cmmdc (%ld,%ld)=%ld\n",a,b,x);
printf("Cmmmc (%ld,%ld)=%ld\n",a,b,a/x*b);

```

5.3. Instrucțiunea do ... while

Are următoarea structură:

```

do
{
    /* grup de instructiuni */
}
while (conditie);

```

Semnificația instrucțiunii este: se execută *grupul de instrucțiuni* cât timp *condiția* este adevărată. Părăsirea ciclării se poate face forțat de asemenea cu *break*.

Instrucțiunea *do...while* din C este echivalentă instrucțiunii repetitive *repetă...până când* din pseudo-cod, deosebirea esențială constă în faptul că la instrucțiunea din C ciclarea se oprește când condiția devine falsă, pe când la instrucțiunea din pseudo-cod ciclarea se încheie când condiția devine adevărată. Cu alte cuvinte condiția din instrucțiunea *do...while* este negația condiției de la instrucțiunea *repetă...până când*.

Luăm ca exemplu sortarea crescătoare unui șir prin metoda bulelor (*BubbleSort*). Despre tablouri de elemente (vectori, matrici) o să vorbim mai târziu. Ceea ce ne interesează acum, pentru a înțelege exemplul de mai jos, este faptul că indicii vectorilor se dau între paranteze pătrate [], iar în momentul declarării unui vector putem enumera elementele sale între acolade.

```

int ok, n=5, a[5]={4, 2, 5, 7, 0};

do
{
    ok=1;
    for (i=0;i<n-1;i++)
        if (a[i]>a[i+1])
        {
            aux=a[i];
            a[i]=a[i+1];
            a[i+1]=aux;
            ok=0;
        }
}
while (!ok);

```

Rezumat

În C există trei instrucțiuni repetitive: **for**, **while** și **do....while**.

Implementarea instrucțiunii *for* este destul de complexă, în sensul că putem face mai multe inițializări, putem avea o condiție complexă de terminare a ciclării și putem avea mai multe actualizări. Aici se utilizează practic operatorul virgulă.

Instrucțiunea *do....while* este cu test final. Cu ajutorul ei putem implementa o instrucțiune pseudocod *repeat....until*.

Teme

1. Se citește de la tastatură un număr întreg. Să se verifice dacă este prim.
2. Afișați primele n numere naturale prime, unde n este citit de la tastatură.
3. Se citește de la tastatură un număr natural. Să se verifice dacă este palindrom. Un număr natural este palindrom dacă cifrele lui citite de la stânga spre dreapta sunt aceleași cu situația în care le citim de la dreapta spre stânga.
4. De la tastatură se citesc două numere naturale n și k . Să se calculeze și să se afișeze pe ecran valoarea expresiei:

$$S(n, k) = \sum_{i=1}^n i^k.$$

6. Tipul char

Obiective

Ne propunem să vedem unde găsim și care sunt principalele funcții scrise pentru tipul **char** din limbajul C.

Tipul *char* a fost prezentat la capitolul dedicat tipurilor întregi de date. Informația reținută într-o valoare de tip *char* poate fi interpretată în C ca un caracter al tabeli ASCII. De acest lucru o să ne ocupăm în acest capitol.

Definițiile principalelor funcții de lucru cu caractere în C le găsim în fișierul antet *ctype.h*. Dintre aceste funcții o să le prezentăm numai pe cele mai importante și mai utile. Facem observația că toate funcțiile de mai jos returnează o valoare de tip *int* și primesc ca argument un caracter:

1) *isalnum(c)* returnează valoare de adevărat (nenulă) dacă c este un caracter alfa-numeric (litera mică, literă mare sau cifră), altfel se returnează 0.

2) *isalpha(c)* returnează valoare de adevărat (nenulă) dacă c este o literă, altfel se returnează 0.

3) *isdigit(c)* returnează valoare de adevărat (nenulă) dacă c este o cifră, altfel returnează 0.

4) *isxdigit(c)* returnează valoare de adevărat (nenulă) dacă c este cifră hexazecimală (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f), altfel se returnează 0.

5) *islower(c)* returnează valoare de adevărat (nenulă) dacă c este literă mică, altfel returnează 0.

6) *isupper(c)* returnează valoare de adevărat (nenulă) dacă c este literă mare, altfel returnează 0.

7) *tolower(c)* returnează transformarea caracterului c în litera mică dacă este literă mare, altfel se returnează valoarea lui c nemodificată.

8) *toupper(c)* returnează transformarea caracterului *c* în litera mare dacă este literă mică, altfel se returnează valoarea lui *c* nemodificată.

Și în fișierul antet *conio.h* întâlnim câteva funcții legate de tipul *char*:

1) *getch()* citește un caracter de la tastatură, fără ecou pe ecranul monitorului. Funcția returnează caracterul citit fără a-l afișa pe ecran.

2) *getche()* citește un caracter de la tastatură, se returnează caracterul citit după ce este afișat pe ecran (cu ecou).

3) *putch(c)* afișează pe ecran caracterul *c* primit ca argument de funcție. Se ține cont de eventuala fereastră text definită cu funcția *window*.

4) *kbhit()* verifică dacă în buffer-ul de intrare de la tastatură există caractere. Cu alte cuvinte se verifică dacă s-a apăsător un buton de la tastatură. Dacă buffer-ul este nevid, atunci se returnează o valoare nenulă.

Programul următor afișează codul ASCII pentru o literă mică preluată de la tastatură:

```
char c;
c=getch();
if (97<=c && c<='z')
    printf("Litera mica %c cu codul ASCII %d!\n",c,c);
while (kbhit()) getch(); /* golire buffer */
printf("Apasati Escape pentru a parasi programul");
do c=getch();
while (c!=27 && c!=13); /* se paraseste daca se apasa
                        Esc sau Enter */
if (c==27) exit(0); /* Functia exit intrerupe executia
                    programului */
```

Rezumat

Tipul caracter coexistă în C cu tipul întreg de date. Funcțiile special scrise pentru tipul caracter au antetele definite în fișierul **ctype.h**.

Temă

Să se afișeze pe ecran cele 256 caractere ale tablei ASCII.

7. Pointeri. Tablouri de elemente.

Obiective

Ne propunem să înțelegem în acest capitol cea mai importantă noțiune a limbajului C. Este vorba de pointer.

Pentru a stăpâni modul de funcționare al limbajului C, trebuie înțeleasă foarte bine noțiunea de pointer.

O variabilă de tip pointer reține o adresă de memorie la care se află o informație (un caracter, un întreg, un șir etc.).

O variabilă de tip pointer se declară sub forma: *tip *numepointer*. De exemplu, o variabilă de tip pointer către tipul *int* se declară astfel: *int *pint*. Variabila *pint* va reține adresa de memorie la care se află stocată o dată de tip *int*.

Revenind la declarația generală *tip *numepointer*, *numepointer* este o variabilă de tipul *tip** și memorează o adresă de memorie, la care este reținută o informație de tipul *tip*. O adresă se memorează pe 4 octeți și în consecință *sizeof(tip*)* este 4.

Putem aplica operatorul *** unei valori de tip pointer pentru a afla valoarea care se află la adresa reținută de pointer. Astfel, **numepointer* reprezintă valoarea de tipul *tip* aflată în memorie la adresa *numepointer*.

Adresa la care se află zona de memorie rezervată unei variabile se poate afla folosind operatorul *&*. Cu alte cuvinte, operatorul *&* este inversul operatorului ***. În consecință, pentru o variabilă *var* de tipul *tip* are sens o atribuire de forma *numepointer=&var*. De asemenea, se poate face atribuirea *var=*numepointer*. Operatorul *** anulează pe *&* și *&* anulează pe ***, adică **&var* este totuna cu *var*, iar *&*numepointer* este același lucru cu *numepointer*.

Pentru valori de tip pointer funcționează operatorii: *=*, *+*, *-*, *+=*, *-=*, *++*, *--* și *!*. De exemplu putem scrie *numepointer+=10*, sau *numepointer=&var-1*.

După aplicarea operatorului *+=* pointerului *numepointer* sub forma *numepointer+=10*, pointerul va reține adresa aflată cu 10 căsuțe mai la dreapta în memorie față de adresa inițială. O căsuță de memorie are *sizeof(tip)* octeți, unde *tip* este tipul de dată către care pointează *numepointer*. Cu alte cuvinte, în urma atribuirii *numepointer+=10*, variabila *numepointer* va reține adresa de memorie aflată cu *10*sizeof(tip)* octeți la dreapta adresei de memorie inițiale din *numepointer*.

De reținut este și faptul că pentru valori de tip pointer nu funcționează operatorii: ***, */*, *%*, adică pointerii nu se pot înmulți, împărți cu alte valori.

7.1. Alocarea statică a memoriei

În C există posibilitatea alocării memoriei atât mod static, cât și dinamic. Prin alocare statică înțelegem faptul că memoria este alocată automat încă de la pornirea execuției instrucțiunilor unei funcții, iar eliberarea memoriei se face tot automat la părăsirea funcției. Cu alte cuvinte, când se intră într-o funcție (inclusiv funcția *main*) se face alocarea statică a memoriei, iar eliberarea se face la părăsirea acelei funcții. Lungimea zonei de memorie alocate static este constantă, adică de fiecare dată când se intră cu execuția într-o funcție, se alocă automat același număr de octeți.

Spre deosebire de alocarea statică, în cazul alocării dinamice stabilim noi momentul alocării memoriei și lungimea zonei de memorie alocate. De asemenea, putem decide momentul în care să se facă eliberarea memoriei.

Pentru un vector putem alocă memorie în mod static sub forma *tip numevector[lungimev]*, unde *lungimev* este o constantă numerică care reprezintă numărul de elemente (căsuțe de memorie) al vectorului.

Dăm câteva exemple de alocări statice de memorie pentru vectori:

```
# define N 100
....
float a[10],b[N];
int i,n,vect[N+1];
```

Elementele unui vector în C au indici între 0 și *lungimev-1*. De exemplu, vectorul *a* definit mai sus are elementele: *a[0]*, *a[1]*, ..., *a[9]*.

Dacă se trece de capetele 0 și *lungimev-1* ale vectorului *a*, în C nu se semnalează în general eroare. Astfel se poate accesa elementul *a[-1]* (zona de memorie aflată cu o căsuță în

stânga lui $a[0]$) sau se poate accesa $a[lungime]$. Acest lucru poate conduce la blocarea programului sau la alterarea altor date reținute în memorie la adresele respective, pe care le-am accesat greșit!

În declarația *numevector[lungimev]*, *numevector* este un pointer care reține adresa de început a zonei de memorie alocate static pentru vector (elementele vectorului se rețin în căsuțe succesive de memorie). Cu alte cuvinte, pointerul *a* reține adresa la care se află memorat primul element $a[0]$ al vectorului *a*, iar $a+1$ este adresa la care se află elementul $a[1]$, $a+2$ este adresa lui $a[2]$ etc. De asemenea, $*a$ este tot una cu $a[0]$, $*(a+1)$ este $a[1]$ etc. În general, $a[k]$ reprezintă o scriere simplificată pentru $*(a+k)$.

7.2. Alocarea dinamică a memoriei

Alocarea dinamică a memoriei în C se poate face cu ajutorul funcțiilor *malloc*, *calloc* și *realloc*, ale căror definiții le găsim în fișierul antet *malloc.h*. Cele 3 funcții pentru alocare dinamică returnează toate tipul pointer către *void*, adică *void**. De aceea, în momentul folosirii acestor funcții trebuie făcută o conversie de tip. Oricare dintre cele trei funcții de alocare returnează valoarea *NULL* (constanta *NULL* are valoarea 0) în cazul în care alocarea de memorie nu s-a putut face (nu este suficientă memorie liberă sau lungimea zonei de memorie ce se dorește a fi alocată nu este validă, adică este negativă).

Funcția *malloc* primește un parametru de tip întreg, care reprezintă numărul de octeți de memorie ce se dorește a fi alocați. De exemplu, pentru un vector de numere întregi scurte trebuie să definim un pointer către tipul *int*, pointer care va reține adresa către zona de memorie alocată:

```
int i,*a,n,m;
printf("Dati lungimea vectorului: ");
scanf("%d",&n);
a=(int*)malloc(n*sizeof(int)); /* alocare memorie pentru n
                                elemente de tip int */

if (a==NULL)
{
    perror("Memorie insuficienta!"); /* mesaj eroare */
    exit(1); /* parasire program cu cod de eroare */
}
for (i=0;i<n;i++)
{
    printf("a[%d]=",i);
    scanf("%d",&a[i]);
}
```

Funcția *malloc* returnează tipul *void**, care trebuie convertit în exemplul de mai sus la tipul *int**, adică la tipul variabilei *a*.

Funcția *calloc* are doi parametri. Primul parametru reprezintă numărul de blocuri ce se alocă, iar al doilea este lungimea unui bloc. Alocarea memoriei pentru vectorul *a* din exemplul de mai sus poate fi rescrisă folosind funcția *calloc* astfel:

```
a=(int*)calloc(n,sizeof(int));
```

Spre deosebire de funcția *malloc*, funcția *calloc* inițializează zona de memorie alocată cu 0, adică toți octeții sunt setați la valoarea 0.

Funcția *realloc* este folosită pentru ajustarea lungimii unei zone de memorie deja alocate, copiind conținutul memoriei anterior alocate dacă este necesar la o nouă adresă. Funcția are doi

parametri. Primul reprezintă adresa zonei de memorie pentru care se dorește să se facă realocarea, iar al doilea este noua lungime (în octeți) a memoriei ce se vrea a fi realocată.

Dacă lungimea memoriei realocate este mai mică sau egală decât lungimea zonei de memorie inițiale, atunci adresa rămâne nemodificată (și este returnată), eventuala diferență de memorie se eliberează.

Dacă memoria realocată este mai mare decât cea inițială, atunci se alocă o nouă zonă de memorie în care se copiază informația din zona inițială, după care prima zonă de memorie se eliberează, în final returnându-se noua adresă. În exemplul următor realocăm memorie pentru un vector *a* cu elemente întregi:

```
printf("Dati noua lungime a vectorului: ");
scanf("%d", &m);
a=(int*) realloc(a,m*sizeof(int));
if (a==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
```

Eliberarea memoriei alocate cu *malloc*, *calloc* și *realloc* se face cu ajutorul funcției *free*, care primește ca parametru pointerul spre zona de memorie alocată. Pentru exemplele de mai sus eliberarea memoriei pentru vectorul *a* se face cu *free(a)*.

Alocarea memoriei pentru o matrice se poate face, de asemenea, atât static cât și dinamic. Varianta statică este: *tip a[nl][nc]*; unde *nl* reprezintă numărul de linii, iar *nc* este numărul de coloane al matricii.

În continuare prezentăm două moduri în care se poate face alocare dinamică de memorie pentru o matrice *a* de valori reale (*float*) de dimensiuni *m* și *n*. În C, o matrice alocată dinamic este un vector (de lungime *m*) de vectori (fiecare de lungime *n*). Pentru aceasta trebuie să definim variabila *a* care va fi de tipul *float***, adică pointer către pointer către tipul *float*. Pointerul *a* va fi adresa către zona de memorie în care se reține vectorul cu adresele de început ale fiecărei linii ale matricii. Întâi va trebui să alocăm memorie pentru vectorul de adrese de lungime *m*. Apoi vom alocă memorie necesară stocării celor *m* x *n* elemente ale matricii. În final vom face legăturile între vectorul de adrese și zona de memorie unde vor fi stocate elementele matricii. În figura 1 este prezentată schema de alocare dinamică de memorie descrisă:

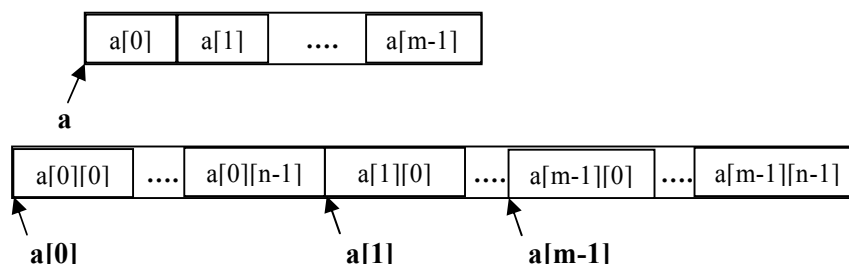


Fig. 1: Prima schemă de alocare dinamică a memoriei pentru o matrice

Codul C de alocare dinamică a memoriei pentru o matrice este:

```
int i,j,m,n;
float **a;
printf("Dati dimensiunile matricii: ");
scanf("%d%d", &m, &n);
a=(float**) calloc(m, sizeof(float*));
```

```

if (a==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
a[0]=(float*)malloc(m*n,sizeof(float));
if (a[0]==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
for (i=1;i<m;i++) a[i]=a[i-1]+n; /* adresele de inceput ale
                                liniilor */

....
free(a[0]); /* eliberarea memoriei ocupata de matrice */
free(a);

```

Întâi se alocă memorie pentru vectorul de adrese al liniilor (a este un vector de elemente de tipul $float^*$). Apoi se alocă memorie necesară stocării tuturor elementelor matricii a (se alocă $m \times n$ blocuri de lungime $sizeof(float)$), iar adresa către zona alocată se depune în $a[0]$. $a[1]$ va fi adresa către zona de memorie aflată cu n căsuțe (fiecare de lungime $sizeof(float)$) după $a[0]$ etc. Cu alte cuvinte la adresa $a[0]$ se găsesc elementele primei linii, apoi în continuare elementele celei de-a doua linii care încep să fie memorate de la adresa $a[1]$, apoi a treia linie la adresa $a[2]$ ș.a.m.d., iar la sfârșit la adresa $a[n-1]$ avem elementele ultimei linii.

Eliberarea zonei de memorie ocupate de matricea a se face în doi pași: întâi se eliberează vectorul cu cele $m \times n$ elemente de tip $float$ (aflat la adresa $a[0]$), iar apoi se eliberează memoria ocupată cu vectorul de adrese (vector aflat la adresa a).

Dezavantajul alocării dinamice de mai sus constă în faptul că se încearcă alocarea unei zone continue de memorie de lungime totală $m \times n \times sizeof(float)$ și s-ar putea să nu dispunem de o zonă continuă de memorie liberă de o asemenea dimensiune. Acest dezavantaj în prezent nu poate fi considerat major, pentru că astăzi calculatoarele dispun de memorie RAM de capacități mari. Marele avantaj al alocării dinamice de mai sus este dat de viteza mare de execuție datorată faptului că se fac numai două alocări și tot atâtea eliberări de memorie.

O alternativă la alocarea de mai sus este alocarea separată a memoriei pentru fiecare linie a matricii:

```

int i,j,m,n;
float **a;
printf("Dati dimensiunile matricii: ");
scanf("%d%d",&m,&n);
a = (float**)calloc(m,sizeof(float*));
if (a==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
for (i=0;i<m;i++)
{
    a[i] = (float*)calloc(n,sizeof(float));
    if (a[i]==NULL)
    {
        printf("Memorie insuficienta!");
        exit(1);
    }
}

```

```

    }
}
....
for (i=0;i<m;i++) free(a[i]);  /* eliberarea memoriei */
free(a);

```

În exemplul de mai sus se fac $m+1$ alocări de memorie pentru $m+1$ vectori. Primul vector alocat (de lungime m), ca și la prima variantă de alocare a memoriei pentru o matrice, va reține adresele către fiecare din cele m linii ale matricii. Apoi se încearcă m alocări de vectori de lungime n , vectorii cu elementele fiecărei linii a matricii. Astfel, $a[0]$, $a[1]$, ..., $a[m-1]$ sunt adresele către fiecare linie obținute în urma alocărilor. Eliberarea memoriei alocate pentru matricea a se face tot în $m+1$ pași: întâi se eliberează cele m zone de memorie în care sunt reținute elementele matricii și în final se eliberează zona de memorie ocupată de vectorul cu adresele de început ale liniilor.

Să facem observația că în urma alocării statice sau a oricărei alocări dinamice de mai sus, referirea la elementul matricii a aflat pe linia i și coloana j se face sub forma $a[i][j]$.

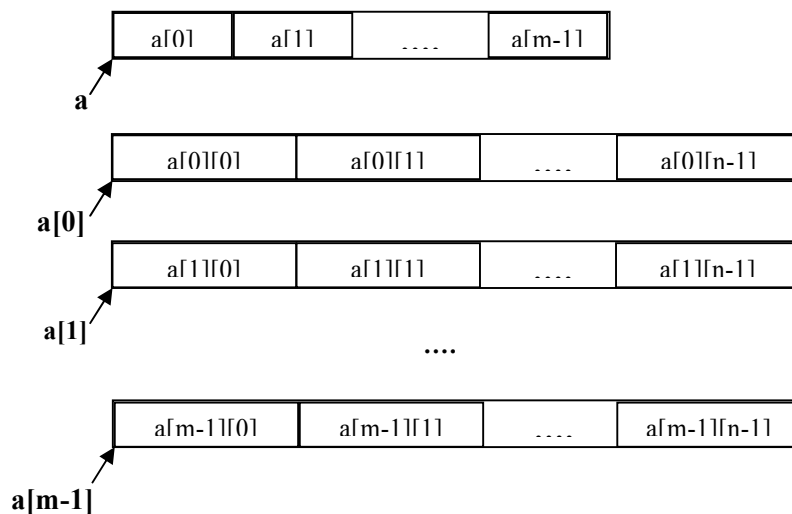


Fig. 2: A doua schemă de alocare a memoriei pentru o matrice

În continuare dăm o secvență de cod în care citim elementele unei matrici alocate static sau dinamic:

```

for (i=0;i<m;i++)
    for (j=0;j<n;j++)
    {
        printf("a[%d,%d]=", i+1, j+1);
        scanf("%f", &a[i][j]);
    }

```

Să facem în final câteva observații:

- 1) În mod similar se poate alocă dinamic memorie pentru matrici de ordin superior (3, 4 etc.). Lășăm ca exercițiu cititorului scrierea codului C de alocare dinamică și eliberare a memoriei pentru o matrice tridimensională.
- 2) Procedând în mod asemănător cu alocarea dinamică a unei matrici putem alocă memorie pentru vectori de vectori, în sensul că fiecare dintre vectorii de la adresele $a[0]$, $a[1]$, ... , $a[m-1]$ pot avea lungimi diferite:

```
int i,j,m,*n;
float **a;

printf("Dati numarul de vectori: ");
scanf("%d",&m);
n=(int*)calloc(m,sizeof(int));
if (n==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
printf("Dati numarul de elemente al fiecarui vector:\n");
for (i=0;i<m;i++)
{
    printf("n[%d]=",i+1);
    scanf("%d",&n[i]);
}
a=(float**)calloc(m,sizeof(float*));
if (a==NULL)
{
    printf("Memorie insuficienta!");
    exit(1);
}
for (i=0;i<m;i++)
{
    a[i]=(float*)calloc(n[i],sizeof(float));
    if (a[i]==NULL)
    {
        printf("Memorie insuficienta!");
        exit(1);
    }
}
printf("Dati elementele vectorilor:\n");
for (i=0;i<m;i++)
    for (j=0;j<n[i];j++)
    {
        printf("a[%d,%d]=",i+1,j+1);
        scanf("%f",&a[i][j]);
    }
....
for (i=0;i<m;i++) free(a[i]); /* eliberarea memoriei */
free(a);
free(n);
```

Rezumat

Un pointer reține o adresă de memorie la care se află memorată o anumită dată (un întreg, un număr real etc.). Pentru a obține valoarea de la adresa reținută în pointer se aplică operatorul * pointerului. Pentru a obține adresa la care se află memorie alocată pentru o variabilă folosim operatorul &.

Alocarea dinamică a memoriei în C se face cu ajutorul funcțiilor **calloc**, **malloc** și **realloc**, ale căror antete se găsesc în fișierul **malloc.h**. Eliberarea memoriei alocate dinamic în C se face cu ajutorul funcției **free**.

În C putem alocă memorie pentru un tablou (vector, matrice etc.) atât în mod static cât și dinamic (în timpul rulării aplicației). Alocarea dinamică a memoriei pentru o matrice bidimensională se face în două etape: întâi alocăm memorie pentru a reține adresele de început ale liniilor matricei și apoi alocăm memorie pentru a reține elementele matricei. În consecință, eliberarea memoriei se face tot în două etape, dar în ordine inversă: eliberăm întâi memoria alocată anterior dinamic pentru a reține elementele matricei și apoi eliberăm memoria ocupată de vectorul de adrese de început a liniilor matricei.

Teme

1. De la tastatură se citește un număr natural nenul n . Să se aloce dinamic memorie pentru o matrice triunghiulară de numere reale (prima linie are un element, a doua are două elemente, a treia trei elemente ș.a.m.d.). Să se citească de la tastatură elementele matricei, să se calculeze și să se afișeze mediile aritmetice pe linii. În final se va elibera memoria alocată dinamic pentru stocarea matricei.
2. De la tastatură se citesc trei numere naturale nenule m , n și p . Să se aloce dinamic memorie pentru o matrice tridimensională de dimensiuni m , n și p . Să se citească de la tastatură elementele matricei. Să se găsească cel mai mare și cel mai mic element al matricei. În final să se elibereze memoria alocată dinamic pentru stocarea matricei.
3. Construieți dinamic un vector în care să depuneți primele n numere naturale prime, unde n este citit de la tastatură.
4. De la tastatură se citesc două numere naturale nenule m și n . Să se aloce dinamic memorie pentru o matrice de dimensiuni m și n . Să se citească de la tastatură elementele matricei. Să se depună elementele matricei luate în spirală într-un vector alocat dinamic. În final să se elibereze toate zonele de memorie alocate dinamic.

De exemplu, pentru matricea:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

vectorul rezultat este (1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7).

8. Funcții în C

Obiective

Ne propunem să vedem cum se redactează o funcție în C și cum se transmit parametrii funcției.

În C, C++ și Java (limbajele ce au la bază standardul ANSI C) ca subprograme nu avem decât funcții, o procedură putând fi implementată ca o funcție care returnează tipul *void* (vid).

În C, o funcție care returnează alt tip decât *void* poate fi apelată atât ca o funcție (în cadrul unei expresii), cât și ca o procedură (ignorându-se valoarea returnată). De exemplu, funcția *printf* returnează valoarea *int*, care reprezintă numărul de octeți care s-au afișat pe ecran. De cele mai multe ori (după cum am văzut și în exemplele de mai sus) funcția *printf* este apelată ca o procedură, ignorându-se valoarea returnată.

După cum am mai spus și în primul capitol, funcțiile pot fi descrise atât la începutul programului, deasupra funcției principale, cât și după aceasta, situație în care definițiile funcțiilor trebuie totuși să apară deasupra funcției principale.

Structura unei funcții în C este următoarea:

```
tip nume_functie(argumente)
{
    /* corpul functiei (instructiuni) */
}
```

Definiția funcției începe cu *tip*, care este tipul valorii pe care o returnează funcția. Dacă *tip* nu este *void*, atunci funcția va conține de regulă cel puțin o instrucțiune *return expresie*. Dacă execuția programului ajunge la o astfel de instrucțiune, atunci se evaluează expresia a cărei valoare trebuie să fie de tipul *tip* sau unul compatibil și valoarea obținută se returnează, execuția programului continuând cu instrucțiunile de după locul în care a fost apelată funcția.

Dacă o funcție ce returnează o valoare de un tip diferit de *void* nu se termină cu o instrucțiune *return*, atunci la compilare vom obține mesajul de atenționare *Warning 5: Function should return a value*.

Spre exemplificare vom scrie o funcție care returnează ca valoare a funcției media aritmetică a două numere reale primite ca parametri:

```
float medie(float x, float y)
{
    return (x+y)/2;
}

void main()
{
    float x, y;
    printf("Dati doua numere reale: ");
    scanf("%f%f", &x, &y);
    printf("Media aritmetica: %f", medie(x, y)); /* apel
                                                functie */
}
```

Rescriem programul de mai sus cu descrierea funcției *medie* după funcția principală:

```
float medie(float, float);

void main()
{
    float x, y;
    printf("Dati doua numere reale: ");
    scanf("%f%f", &x, &y);
    printf("Media este: %f", medie(x, y));
}
```



```

}

float medie(float x, float y)
{
    return (x+y)/2;
}

```

După cum se poate observa mai sus, când descriem funcția *medie* după funcția principală, la definiție nu este obligatoriu să dăm numele argumentelor funcției, iar definiția se încheie cu caracterul punct și virgulă.

În C parametrii ce se returnează din funcție (variabilele de ieșire) se transmit prin adresă. În acest sens dăm un exemplu în care media aritmetică este returnată printre parametrii funcției:

```

void medie2(float x, float y, float *m)
{
    *m=(x+y)/2;
}

void main()
{
    float x,y,med;
    printf("Dati doua numere reale: ");
    scanf("%f%f",&x,&y);
    medie2(x,y,&med); /* apel functie */
    printf("Media este: %f",med);
}

```

La apelul unei funcții valorile parametrilor de intrare, cei transmiși prin valoare (așa cum este cazul parametrilor x și y), sunt copiate în zone de memorie noi și de aceea, dacă le modificăm, la părăsirea funcției, valorile modificate se pierd, zonele de memorie alocate pentru acești parametri în funcție fiind eliberate.

Parametrul m al funcției *medie2* este transmis prin adresă. Mai exact, în funcție se transmite adresa la care este alocată memorie pentru variabila *med* din funcția principală. În funcția *medie2* se modifică valoarea de la adresa m , adresă care coincide cu cea a variabilei *med* și de aceea valoarea calculată rămâne în variabila *med* după ce se părăsește funcția.

Dacă vrem ca o variabilă transmisă prin adresă să nu poată fi modificată, punem în fața ei cuvântul rezervat **const**. În general, dacă se încearcă modificarea unei variabile declarate cu *const* (a unei constante), atunci se semnalează eroare la compilare.

Variabilele de tip tablou sunt pointeri, de aceea valorile de la adresele indicate de ele se transmit prin adresă și eventualele modificări în interiorul funcției asupra elementelor tabloului păstrându-se și la părăsirea funcției.

În continuare prezentăm o funcție care calculează media aritmetică a elementelor unui șir de numere reale:

```

float mediesir(int n, const float *a)
{
    int i;
    float s=0;

    for (i=0;i<n;i++) s+=a[i];
    return s/n;
}

```

```

void main()
{
    int i,n;
    float a[100];

    printf("Dati numarul de elemente al sirului: ");
    scanf("%d",&n);
    puts("Dati elementele sirului:");
    for (i=0;i<n;i++)
    {
        printf("a[%d]=",i);
        scanf("%f",&a[i]);
    }
    printf("Media aritmetica: %f",mediesir(n,a));
}

```

Trebuie să fim atenți pentru a înțelege corect modul de transmitere al parametrilor prin adresă. Dacă modificăm valoarea de la adresa reținută în pointerul - parametru al funcției, noua valoare va fi vizibilă și în variabila transmisă prin adresă, așa cum este cazul aplicației scrise pentru funcția *medie2*. Ceea ce trebuie să înțelegem este că un pointer se transmite prin valoare, ca orice variabilă. De aceea, dacă modificăm adresa (nu valoarea de la adresa reținută de pointer !) memorată în pointer, noua adresă nu se returnează ! Cea mai des întâlnită situație în care adresa reținută în pointer se modifică este atunci când alocăm memorie într-o funcție și vrem să returnăm adresa spre zona nou alocată. În această situație pointerul va trebui transmis prin adresă ! Spre exemplificare, scriem o funcție care alocă memorie pentru un vector de elemente reale. Adresa spre zona de memorie alocată se returnează prin adresă, ca parametru al funcției:

```

#include<stdio.h>
#include<alloc.h>

void AlocVector1(float *a,int n) /* gresit */
{
    a=(float*)malloc(n*sizeof(float));
}

void AlocVector2(float **a,int n) /* corect */
{
    *a=(float*)malloc(n*sizeof(float));
}

float* AlocVector3(int n) /* adresa noua returnata ca val. a fct. */
{
    return (float*)malloc(n*sizeof(float));
}

int main()
{
    int n;
    float *a;

    printf("n=");
    scanf("%d",&n);
    AlocVector1(a,n); /* nu se returneaza adresa noua */
}

```

```

AlocVector2(&a,n); /* pointerul a transmis prin adresa */
a=AlocVector3(n);
if (a==NULL)
{
    perror("Memorie insuficienta!");
    return 1; /* parasire program cu cod de eroare */
}
/* .... */
free(a);
return 0; /* executia programul s-a incheiat cu succes */
}

```

Când într-o funcție se modifică și se dorește a fi returnată o singură adresă, din comoditate și pentru a fi mai ușor de utilizat și de înțeles, noua adresă se returnează ca valoare a funcției. Așa este cazul funcțiilor de alocare dinamică de memorie din fișierul antet *alloc.h* (*malloc*, *calloc*, *realloc*). Fiecare dintre aceste funcții returnează adresa zonei de memorie nou alocate ca valoare a funcției. Rescriem și noi funcția de alocare dinamică de memorie pentru un vector de elemente reale cu returnarea adresei spre noua zonă de memorie ca valoare a funcției:

```

#include<stdio.h>
#include<alloc.h>

float* AlocVector3(int n) /* adresa noua returnata ca val. a fct. */
{
    return (float*)malloc(n*sizeof(float));
}

int main()
{
    int n;
    float *a;

    printf("n=");
    scanf("%d",&n);
    a=AlocVector3(n);
    if (a==NULL)
    {
        perror("Memorie insuficienta!");
        return 1;
    }
    /* .... */
    free(a);
    return 0;
}

```

Rezumat

Definiția unei funcții în C începe cu tipul returnat de către funcție, urmat de numele funcției și de lista de argumente dată între paranteze. Dacă o funcție are tipul returnat **void**, atunci ea nu returnează nici o valoare și se apelează practic ca o procedură.

Parametrii de intrare în funcție se transmit prin valoare, iar cei de ieșire se transmit prin adresă.

Teme

1. Scrieți o funcție pentru căutare binară a unei valori într-un șir de numere reale sortat crescător.
2. Scrieți o funcție pentru căutare secvențială a unei valori într-un șir de numere întregi. Funcția returnează numărul de apariții a unei valori întregi în șir.
3. Scrieți o funcție care sortează crescător un șir de numere reale folosind metoda BubbleSort.
4. Aceași problemă folosind metoda QuickSort.
5. Aceași problemă folosind metoda MergeSort.

9. String-uri

Obiective

Ne propunem să studiem modul în care se memorează și cum se lucrează cu un string în limbajul C.

În C, un șir de caractere care se termină cu caracterul *NULL* sau `'\0'` sau pur și simplu *0* (de unde și denumirea de șir de caractere *NULL* terminat) este echivalentul noțiunii de *string* din alte limbaje (cum ar fi de exemplu limbajul Pascal). În consecință, în C o variabilă string este de fapt un pointer de tipul *char*. Fiind vorba de un șir, alocarea memoriei pentru un *string* se poate face atât static cât și dinamic.

În fișierul antet *stdio.h* avem definite funcțiile *gets* și *puts* pentru citirea unui string de la tastatură, respectiv afișarea pe ecran a unui string. Ambele funcții primesc ca parametru o variabilă de tip pointer către tipul *char*.

Funcția *gets*, spre deosebire de *scanf* poate citi string-uri care conțin spații. În cazul funcției *scanf* se consideră încheiată introducerea unei valori pentru o variabilă când se întâlnește caracterul spațiu sau unde s-a apăsă *Enter*. De exemplu, dacă se citește *Popescu Ioan* pentru un string *s* de la tastatură cu *gets(s)*, atunci la adresa *s* se depun caracterele *Popescu Ioan'\0'*. Dacă citirea se face însă cu *scanf("%d",s)*, atunci la adresa *s* se depun caracterele *Popescu'\0'* (citirea lui *s* se încheie la apariția spațiului).

Dacă se afișează un string *s* cu *puts(s)*; atunci după afișarea șirului de la adresa *s* se face salt la începutul liniei următoare, ceea ce nu se întâmplă dacă afișarea se face cu *printf("%s",s)*.

Fișierul antet *string.h* conține definițiile funcțiilor C de lucru cu string-uri:

- 1) *strcat(s1, s2)* la sfârșitul string-ului *s1* se adaugă *s2* și caracterul `'\0'`.
- 2) *strchr(s,c)* returnează adresa de memorie către prima apariție a caracterului *c* în string-ul *s*. Dacă nu există caracterul în string, se returnează adresa nulă (*NULL*).
- 3) *strcmp(s1,s2)* returnează o valoare întreagă care dacă este *0*, înseamnă că șirurile sunt identice ca și conținut. Dacă valoarea returnată este negativă, atunci înseamnă că string-ul *s1* este înaintea lui *s2* din punct de vedere lexicografic (alfabetic). Dacă însă valoarea returnată este pozitivă, atunci înseamnă că string-ul *s1* este după *s2* din punct de vedere lexicografic.
- 4) *stricmp(s1,s2)* face același lucru ca și *strcmp*, cu deosebirea că nu se face distincție între litere mari și mici.
- 5) *strncmp(s1,s2,n)* realizează o comparație între primele cel mult *n* caractere ale string-urilor *s1* și *s2*.
- 6) *strncmpi(s1,s2,n)* realizează o comparație între primele cel mult *n* caractere ale string-urilor *s1* și *s2* fără a se face distincție între litere mari și mici.

- 7) *strcpy(s1,s2)* copiază caracterele de la adresa *s2* (până se întâlnește inclusiv caracterul ‘\0’) la adresa *s1*. Cu alte cuvinte, se copiază conținutul string-ului *s2* peste string-ul *s1*.
- 8) *strlen(s)* returnează lungimea string-ului *s* (numărul de caractere până la ‘\0’).
- 9) *strlwr(s)* transformă literele mari din string-ul *s* în litere mici.
- 10) *strupr(s)* transformă literele mici din string-ul *s* în litere mari.
- 11) *strncat(s1,s2,n)* adaugă la sfârșitul string-ului *s1* cel mult *n* caractere din string-ul *s2*, după care se adaugă caracterul ‘\0’.
- 12) *strncpy(s1,s2,n)* copiază cel mult *n* caractere de la începutul string-ului *s2* la adresa *s1*, după care se adaugă la sfârșit caracterul ‘\0’.
- 13) *strnset(s,c,n)* setează primele *n* caractere ale string-ului *s* la valoarea primită în parametrul *c*.
- 14) *strpbrk(s1,s2)* returnează adresa de memorie la care apare prima dată un caracter din string-ul *s2* în string-ul *s1*.
- 15) *strchr(s,c)* returnează adresa la care apare ultima dată caracterul *c* în string-ul *s*.
- 16) *strrev(s)* inversează caracterele string-ului *s* (primul se schimbă cu ultimul, al doilea cu penultimul etc.).
- 17) *strset(s,c)* setează toate caracterele string-ului *s* (până la ‘\0’) la valoarea primită în parametrul *c* de tip *char*.
- 18) *strstr(s1,s2)* returnează adresa de memorie la care apare prima dată string-ul *s2* în *s1*. Dacă *s2* nu apare în *s1*, atunci se returnează adresa nulă *NULL*.
- 19) *strxfrm(s1,s2,n)* înlocuiește primele cel mult *n* caractere ale string-ului *s1* cu primele cel mult *n* caractere din string-ul *s2*.

Ca aplicații pentru string-uri propunem:

1. O funcție care returnează printre parametri ei toate pozițiile pe care apare un string *s2* într-un string *s1*.
2. Funcție pentru înlocuirea primei apariții în *s1* a string-ului *s2* cu string-ul *s3*.
3. Funcție pentru înlocuirea tuturor aparițiilor lui *s2* în *s1* cu string-ul *s3*.

Pentru aceste aplicații vom folosi funcții din fișierul antet *string.h*, dar va trebui și să lucrăm direct cu adrese de memorie și asta din cauză că un string este de fapt un pointer către tipul *char*.

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

int strmfnd(char *s1,char *s2,int *a) /* cautarea tuturor aparitiilor */
{
    char *s,*poz;
    int n=0;
    s=s1;                /* s ia adresa la care se afla s1 */
    poz=strstr(s,s2);    /* cautare daca s2 apare in s */
    while (poz!=NULL)    /* daca s2 apare in s */
    {
        a[n]=poz-s1+1;    /* se memorează pozitia aparitiei */
        s=poz+strlen(s2);
        n++;
        poz=strstr(s,s2); /* se cauta o noua aparitie */
    }
}
```

```

        return n; /* se returneaza numarul de aparitii */
    }

int strreplacel(char*s1,char*s2,char*s3) /* inlocuirea primei aparitii */
{
    char *poz,*s4;
    long l;
    poz=strstr(s1,s2); /* se cauta prima aparitie a lui s2 in s1 */
    if (poz!=NULL)      /* daca s2 apare in s */
    {
        l=poz-s1; /* l este pozitia aparitiei (poate fi si 0) */
        s4=(char*)malloc(strlen(s1)+1); /* aloc. mem. pt s4 */
        strcpy(s4,s1); /* se copiaza s1 in s4 */
        strncpy(s1,s4,l); /* se copiaza in s1 l caract din s4 */
        s1[l]='\0'; /* se transforma s1 in string */
        strcat(s1,s3); /* se adauga s3 la sfarsitul lui s1 */
        strcat(s1,poz+strlen(s2)); /* adaugare la sf. lui s1 */
        free(s4); /* ramase in string-ul initial dupa s2 */
        return l+1; /* se returneaza pozitia inlocuirii */
    }
    return 0;
}

int strreplace(char*s1,char*s2,char*s3) /* inloc. tuturor aparitiilor */
{
    int n=0;
    char *poz;
    poz=strstr(s1,s2); /* se cauta s2 in s1 */
    while (poz!=NULL) /* daca exista s2 in s1 */
    {
        n++;
        strreplacel(s1,s2,s3); /* se inloc. s2 cu s3 in s1 */
        poz=strstr(s1,s2); /* se cauta din nou s2 in s1 */
    }
    return n; /* se returneaza numarul de inlocuiri efectuate */
}

void main(void) /* testare functii */
{
    char s1[100],s2[100],s3[100];
    int i,n,a[20];
    strcpy(s1,"Acesta este un text si numai unul !");
    strcpy(s2,"un"); /* string-ul care se inlocuieste */
    strcpy(s3,"1"); /* string-ul cu care se face inlocuirea */
    n=strmfind(s1,s2,a);
    if (n)
    {
        printf("'s' apare in 's' de %d ori, pe poz.:\n",s2,s1,n);
        for (i=0;i<n;i++) printf("%d ",a[i]);
        puts("\n");
        printf("String-ul dupa inloc. lui 's' cu 's':\n",s2,s3);
        strreplace(s1,s2,s3);
        puts(s1);
    }
}

```

```

    }
    else printf("'%' nu apare in '%s'\n",s2,s1);
    getch();
}

```

Rezumat

String-ul se memorează în C într-un șir de caractere obișnuit. Caracterul ‘\0’ (caracterul cu codul ASCII 0, adică NULL) marchează sfârșitul string-ului. De aceea, despre string în C se spune că este un șir NULL terminat.

În fișierul antet **string.h** găsim definițiile funcțiilor de lucru cu string-uri în C.

Teme

1. Folosind funcția *strrev* scrieți o funcție care verifică dacă un număr întreg de tip *unsigned long* primit ca parametru este palindrom. Un număr întreg este palindrom dacă cifrele lui citite de la stânga spre dreapta sunt aceleași cu situația în care sunt citite de la dreapta spre stânga.
2. Scrieți o funcție care returnează numărul de cuvinte dintr-un string primit ca parametru. Cuvintele se consideră a fi grupuri de caractere despărțite prin spații.

10. Structuri

Obiective

Vom studia în acest capitol tipul **struct**, adică tipul compus sau înregistrare.

Cu ajutorul structurilor putem defini tipuri noi de date, mai complexe pornind de la tipuri de date deja existente, care alcătuiesc așa numitele câmpuri ale structurii. O structură se definește astfel:

```

struct nume
{
    tip1 camp1,camp2,...,campm1;
    tip2 camp1,camp2,...,campm2;
    /* .... */
    tipn camp1,camp2,...,campmn;
};

```

După acolada care încheie definirea structurii trebuie neapărat să punem semnul punct și virgulă.

Între acolada } ce încheie definirea structurii și semnul ; (punct și virgulă) putem declara variabile care vor fi evidente de tipul structurii respective.

În C denumirea tipului structură definit în exemplul de mai sus este *struct nume* (denumirea structurii este precedată de cuvântul rezervat *struct*). În C++ nu este obligatoriu ca înainte de *nume* să punem cuvântul rezervat *struct*.

Adesea se preferă (mai ales în C) definirea unei structuri în combinație cu *typedef*. Pentru a ilustra acest lucru definim o structură pentru memorarea unui număr complex ca o pereche de numere reale:

```
typedef struct
{
    double re,im;
} complex;
```

În final pentru a exemplifica modul de lucru cu structuri vom citi datele despre o persoană într-o structură, după care le vom afișa:

```
struct tpers
{
    char nume[50],adresa[100];
    int varsta;
};

typedef struct tpers tpers;

void main()
{
    struct tpers pers;
    puts("Dati datele despre o persoana:");
    printf("Nume:  ");
    gets(pers.nume);
    printf("Adresa: ");
    gets(pers.adresa);
    printf("Varsta: ");
    scanf("%d",&pers.varsta);
    puts("Am citit:");
    printf("Nume:  %s\n",pers.nume);
    printf("Adresa: %s\n",pers.adresa);
    printf("Varsta: %d\n",pers.varsta);
}
```

După cum se poate observa mai sus, referirea la un câmp al unei structuri se face sub forma *var_structura.camp*.

Rezumat

Un tip de date compus pornind de la tipurile de date deja existente în C se definește cu ajutorul cuvântului rezervat **struct**. Accesul la membrul unei structuri se face ajutorul operatorului punct sub forma *variabila_tip_struct.membru*.

Teme

1. De la tastatură citiți stocul unei firme într-un șir de elemente de tip *struct*. În structură se vor reține: denumirea produsului, cantitatea și prețul. Sortați crescător șirul după denumirea produselor, după care afișați tabelar produsele după modelul:

```
-----
|Nr.  | DENUMIRE PRODUS                |Cantitate| Pret   |Valoare|
|crt. |                                |         |        |        |
|-----+-----+-----+-----+-----|
```


	1	Placa de baza		2.00	401.44	802.88
	2	Tastatura		1.00	25.11	25.11
	3	Mouse		22.00	14.00	308.00
	4	Cablu retea		117.50	0.23	27.03
	-----+-----+-----					
	Valoare totala stoc:					1163.02

2. Rezolvați problema 2 de la sfârșitul capitolului 3 folosind un vector de structuri.

11. Pointeri către structuri. Liste înlănțuite.

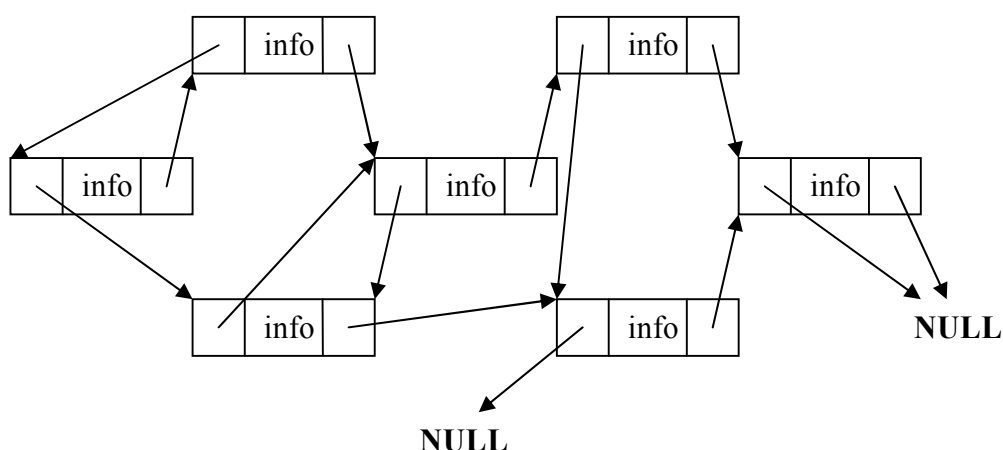
Obiective

Ne propunem să vedem cum se implementează o listă înlănțuită cu ajutorul tipului struct studiat în capitolul anterior.

Pentru început, vom prezenta din punct de vedere teoretic noțiunea de listă înlănțuită și principale structuri de date de tip listă înlănțuită, după care vom vedea cum putem implementa listele înlănțuite cu ajutorul pointerilor către structuri.

O listă înlănțuită este o structură formată dintr-o mulțime de așa numite noduri legate între ele. În fiecare nod se memorează niște informații (numere, caractere, string-uri etc.) și una sau mai multe adrese ale altor noduri din listă sau adrese nule (legăturile). Cele mai folosite liste înlănțuite sunt cu una sau două legături (adrese către alte noduri).

Listele cu două legături se numesc dublu înlănțuite. Fiecare nod memorează informația, iar fiecare dintre cele două legături este către un alt nod ale listei sau este o legătură vidă. Iată un posibil exemplu de listă dublu înlănțuită:



De departe cea mai folosită listă dublu înlănțuită este cea de tip arbore binar. Există un nod special denumit rădăcină. Nici un nod nu are legătura către rădăcină. Pornind din rădăcină se poate ajunge prin intermediul legăturilor la orice alt nod al arborelui. Arborele binar nu are cicluri, adică pornind dintr-un nod x oarecare nu se poate ajunge prin intermediul legăturilor din nou la nodul x .

O listă simplu înlănțuită este formată din noduri ce conțin (fiecare) o informație și o legătură către un alt nod sau o legătură nulă. O listă liniară simplu înlănțuită are proprietatea că

există un nod denumit *cap* care are o legătură nulă sau este legat de un alt nod al listei, care la rândul lui are legătura nulă sau este legat de alt nod etc. Lista are proprietatea că pornind de la nodul *cap* se poate ajunge prin legături la orice alt nod al listei și pentru fiecare nod *x* diferit de *cap* există un singur nod *y* care este legat de *x*.

După modul de introducere și de scoatere a informațiilor într-o listă liniară avem liste de tip stivă sau coadă.

O stivă este o listă liniară de tip LIFO (Last In First Out), adică ultimul intrat – primul ieșit. O stivă (după cum sugerează și numele) ne-o putem imagina cu nodurile stivuite unul peste celălalt. Introducerea unui nou nod se face peste ultimul introdus, iar scoaterea unei informații se face din ultimul nod introdus în stivă. O stivă poate fi memorată printr-o listă liniară simplu înlănțuită în care fiecare nod reține adresa nodului de sub el (ca legătură). Primul nod introdus în stivă are legătura nulă. Pentru o stivă se reține în permanență într-o variabilă de tip pointer (denumită de obicei *cap*) adresa către ultimul nod al stivei. O stivă este goală (vidă) când variabila *cap* reține adresa vidă.

Coadă este tot o listă liniară, dar de tipul FIFO (First In First Out), adică primul intrat – primul ieșit. După cum sugerează și numele, nodurile ni le putem imagina aranjate în linie, unul după altul, introducerea unui nou nod se face după ultimul introdus anterior, de scos, se scoate primul nod introdus în coadă. Coadă poate fi și ea memorată cu ajutorul unei liste liniare simplu înlănțuite în care introducerea unui nod se face într-o parte a listei, iar scoaterea se face din cealaltă parte. Se rețin în permanență adresa (într-o variabilă de tip pointer denumită de obicei *ultim*) ultimului nod introdus în listă și pe cea a primului introdus (într-o variabilă de tip pointer denumită de obicei *prim*). Pentru a funcționa eficient (să putem scoate rapid un nod), fiecare nod al cozii reține adresa nodului din spatele lui (cel introdus imediat după el în coadă). Este evident că primul nod are legătura nulă.

O coadă specială este aceea în care primul nod în loc să aibă legătura nulă, el este legat de ultimul nod al cozii. Această structură de date se numește coadă circulară.

Cu ajutorul pointerilor către structuri putem implementa în C listele înlănțuite. Un pointer către o structură se definește astfel:

```
struct nume
{
    /* definiri de campuri impreuna cu tipurile lor */
};
/* .... */
struct nume *p; /* pointer catre structura */
```

Este important de reținut faptul că referirea la un câmp al unei structuri memorate la adresa dată de pointerul *p* se face sub forma *p->camp*, ceea ce este echivalent cu a scrie *(*p).camp*. Practic, în C pentru pointeri către structuri este introdus operatorul binar special *->* (format din caracterele *minus* și *mai mare*).

Ca aplicație prezentăm un program ce simulează lucrul cu o stivă de caractere folosind pointeri către o structură denumită *struct tstiva*.

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

struct tstiva
{
    char info;
    struct tstiva *leg;
}*cap; /* cap este o variabila globala care va retine adresa capului
        stivei */
```

```

typedef struct tstiva tstiva;

int introducere(char c) /* introducerea unui caracter in stiva */
{
    struct tstiva *p;
    p=(struct tstiva*)malloc(sizeof(tstiva));
    if (p==NULL) return 0;
    p->info=c;
    p->leg=cap;
    cap=p;
    return 1;
}

int scoatere(char *c) /* scoaterea unui caracter din stiva */
{
    struct tstiva *p;
    if (cap==NULL) return 0;
    p=cap;
    cap=p->leg;
    *c=p->info;
    free(p);
    return 1;
}

void afisare() /* afisarea continutului stivei */
{
    struct tstiva *p=cap;
    while (p!=NULL)
    {
        printf("%c ",p->info);
        p=p->leg;
    }
}

void golirestiva()
{
    struct tstiva *p;
    while (cap!=NULL)
    {
        p=cap;
        cap=p->leg;
        free(p);
    }
}

void main()
{
    char c,inf;
    cap=NULL; // setam stiva ca fiind goala
    do
    {
        puts("1) Introducere caracter in stiva");
    }
}

```

```

puts("2) Scoatere ultim caracter in stiva");
puts("3) Afisare continut stiva");
puts("4) Golire stiva\n");
puts("Esc - Parasire program\n");
c=getch();
switch (c)
{
    case '1':
        printf("Dati un caracter: ");
        inf=getche();
        if (!introducere(inf))
            printf("\nMemorie insuficienta");
        else printf("\nAm introdus: %c",inf);
        break;
    case '2':
        if (!scoatere(&inf)) printf("Stiva goala!");
        else printf("Am scos din stiva: %c",inf);
        break;
    case '3':
        printf("Stiva contine: ");
        afisare();
        break;
    case '4':
        golirestiva();
        printf("Am golit stiva!");
        break;
    case 27:
        printf("Apasati o tasta pt. a iesi!");
        break;
    default:
        printf("Apasati 1, 2, 3, 4 sau Escape");
}
getch();
}
while (c!=27);
golirestiva();
}

```

Rezumat

O listă înlănțuită în C se poate implementa folosind pointeri către tipul struct. Accesul la membrul unei structuri aflată la adresa reținută în pointerul **p** se face cu ajutorul operatorului \rightarrow sub forma **p→membru_structura**.

Teme

- 1) Lucrul cu o coadă (în mod asemănător cu programul de mai sus).
- 2) Generarea arborelui binar pornind de la șirurile *stg*, *dr*, și *info*. Vectorii *stg* și *dr* memorează indicii nodului fiu stâng, respectiv fiu drept, iar în *info* se rețin informațiile din fiecare nod.
- 3) Afișarea informației reținute în frunzele unui arbore binar.
- 4) Afișarea informației reținute pe un anumit nivel al unui arbore binar.
- 5) Căutarea unei informații într-un arbore binar.

- 6) Jocul copiilor în cerc (cu o coadă circulară): n copii stau așezați într-un cerc. Se numără m copii consecutivi pornind de la al p -lea. Al m -lea copil iese din cerc. Se continuă număratoarea cu cel ce urmează după copilul ce a părăsit cercul, până când rămâne un singur copil, care este declarat câștigător. Se cere să se afișeze ordinea de părăsire a cercului, precum și câștigătorul.

12. Uniuni

Obiective

Vom studia tipul **union** și utilitatea lui atunci când se apelează instrucțiuni de lucru cu regiștri.

Definirea unei uniuni seamănă foarte mult cu cea de la structuri. În loc de cuvântul *struct* avem însă cuvântul *union*. Iată și un exemplu:

```
union test
{
    char c;
    int i;
    float f;
};
```

Semnul ; (punct și virgulă) se pune de asemenea la sfârșitul definiției. Între acolade {} și ; (punct și virgulă) putem defini variabile de tip uniune.

Similitudinea dintre o uniune și o structură se oprește însă aici.

Pentru a reține o variabilă de tip *union* sunt necesari 4 octeți de memorie. În exemplul de mai sus câmpurile *c*, *i* și *f* ale uniunii se referă toate la aceeași zonă din memorie: *c* se referă la primul octet (din cei patru), *i* se referă la primii doi octeți, iar *f* se referă la toți cei 4 octeți.

Zona de memorie rezervată unei variabile union este interpretată diferit în funcție de tipurile câmpurilor uniunii.

Uniunile sunt folosite în C în special pentru instrucțiuni ce lucrează cu regiștri. În acest scop pentru DOS (în Borland C) este definită uniunea *REGS*:

```
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

BYTEREGS și *WORDREGS* sunt două structuri:

```
struct BYTEREGS
{
    unsigned char al, ah, bl, bh;
    unsigned char cl, ch, dl, dh;
};

struct WORDREGS
{
```

```

    unsigned int ax,bx,cx,dx;
    unsigned int si,si cflag,flags;
};

```

Uniunea REGS este folosită pentru transmiterea parametrilor funcției de bibliotecă *int86* prin intermediul căreia se pot accesa funcțiile ROM-BIOS ale sistemului de calcul. Structura instrucțiunii *int86* este:

```
int86(nr_intrerupere,* in_regs,&out_regs);
```

Avem:

nr_intrerupere este numărul întreruperii apelate
in_regs este valoarea cu care se încarcă regiștrii
out_regs reprezintă valoarea regiștrilor de după apel

Tipul union REGS se găsește definit în fișierul antet *dos.h*.

Ca exemplu pentru utilizarea tipului *union REGS* prezentăm o secvență de program care ascunde cursorul text (se utilizează întreruperea 0x10):

```

union REGS reg;
reg.h.ah=1;
reg.h.ch=0x20;
int86(0x10, &reg, &reg);

```

Pentru a face să reapară cursorul trebuiesc executate:

```

reg.h.ah=1;
reg.h.ch=12;
reg.h.cl=13;
int86(0x10, &reg, &reg);

```

Rezumat

O uniune se definește folosind cuvântul rezervat **union** asemănător cu o structură, diferența esențială dintre o uniune și o structură este că membrii uniunii folosesc în comun aceeași zonă de memorie.

13. Seturi de constante (tipul enum)

Obiective

Ne propunem să vedem cum se pot defini simultan mai multe constante întregi cu ajutorul tipului **enum**.

Un tip de date *enum* se definește asemănător cu o structură sau uniune.

Cu ajutorul tipului **enum** putem defini seturi de constante întregi. Tipul de date *enum* este des folosit pentru a da denumiri diferitelor combinații de biți din alcătuirea unor întregi de formatare, cum ar fi de exemplu biții de formatare ai modului de introducere într-un flux C++

(definiți în fișierul antet *ios.h*): *ios::left*, *ios::right*. Aceste constante sunt puteri ale lui doi pentru că se referă fiecare numai la un anumit bit de pe o anumită poziție din întregul de formatare. Asupra fluxurilor C++ o să revenim la momentul potrivit.

Ca aplicație, definim constante cu nume românești pentru cele 16 culori ale modului text:

```
enum constante_culori
{
    NEGRU=0,
    ALBASTRU=1,
    VERDE=2,
    AZUR=3,
    ROSU=4,
    MOV=5,
    MARO=6,
    GRIINCHIS=7,
    GRIDESCHIS=8,
    ALBASTRUDESCHIS=9,
    VERDEDESCHIS=10,
    CYANDESCHIS=11,
    ROSUDESCHIS=12,
    MOVDESCHIS=13,
    GALBEN=14,
    ALB=15
};
//....
textcolor(GALBEN); /* numai sub DOS */
textbackcolor(ALBASTRU); /* numai sub DOS */
printf("Acesta este un test!");
```

De remarcat este faptul că avem constantele tipului *enum* date între acolade și despărțite prin virgule.

Rezumat

Folosind tipul *enum* putem defini un grup de constante întregi. Constantele împreună cu valorile lor se enumeră între acolade și sunt despărțite prin spații.

Teme

1. Definiți un set de constante întregi cu denumiri de plante. Presupunem că aceste constante reprezintă indicii componentelor unui vector de structuri în care se rețin informații cu privire la diverse plante. Scrieți funcții de actualizare a datelor despre plante folosind constantele pe care le-ați definit (de exemplu puteți scrie o funcție pentru actualizarea prețului unei plante).
2. Aceeași problemă pentru un set de constante întregi cu nume de mărci de autoturisme.

14. Fișiere în C

Obiective

Ne propunem să studiem modul de lucru cu fișiere din C, cum se deschide un fișier, cum se scrie în el, cum se citesc date, cum se face o poziționare în fișier etc.

Pentru a lucra cu un fișier în C se declară o variabilă tip pointer către tipul structură *FILE*. Nu ne interesează în momentul în care lucrăm cu un fișier ce se întâmplă cu câmpurile structurii *FILE*, deoarece avem suficiente funcții care lucrează cu fișiere. De aceea, putem spune despre tipul *FILE* că este un tip abstract de date.

14.1. Funcții de lucru cu fișiere

Funcțiile de lucru cu fluxuri în C sunt definite în *stdio.h*. Așa că atunci când lucrăm cu fișiere în C trebuie să includem acest fișier antet.

Deschiderea fișierului se face folosind funcția *fopen*, care are următoarea definiție (antet):

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

Funcția primește ca argumente două string-uri, primul trebuie să conțină numele fișierului (precedat eventual de cale) și al doilea modul de deschidere al fișierului. Se returnează pointerul către fluxul deschis (pointer către *FILE*). Dacă nu s-a putut deschide fișierul, se returnează adresa nulă (*NULL*).

Deschiderea unui fișier pentru operații de scriere eșuează dacă numele lui (inclusiv calea) conține caractere invalide, fișierul este deja deschis de o altă aplicație sau dacă nu avem drepturi de scriere în calea respectivă.

Un fișier nu poate fi deschis pentru a se citi din el informații decât dacă fișierul există, nu are atributul *hidden* (ascuns) și avem drepturi de citire asupra lui.

Putem deschide un fișier pentru scriere, pentru citire sau pentru adăugare la sfârșit (*append*), în modul binar (fișierul în acest caz e interpretat ca o succesiune de octeți, baiți) sau în modul text (fișierul e interpretat ca o succesiune de linii cu texte despărțite prin caracterele *'\r'* și *'\n'*).

Pentru a deschide un fișier pentru scriere, string-ul din parametrul al doilea al funcției *fopen* trebuie să conțină caracterul *'w'*, pentru a deschide fișierul pentru citire string-ul trebuie să conțină caracterul *'r'*, iar pentru a deschide fișierul pentru adăugări la sfârșit, în string trebuie să apară caracterul *'a'*. Dacă fișierul este deschis cu *'w'*, atunci se va crea un fișier nou, dacă există un fișier cu acest nume, el este mai întâi șters. Pentru a putea fi deschis un fișier cu *'r'*, el trebuie să existe.

Pentru a deschide un fișier în modul binar, string-ul din parametrul al doilea al funcției *fopen* trebuie să conțină caracterul *'b'*. Dacă string-ul nu conține caracterul *'b'* sau conține caracterul *'t'*, atunci el va fi deschis în modul text.

În string-ul ce dă modul de deschidere al fișierului poate să mai apară caracterul *'+'*. Când apare acest caracter pentru un mod de deschidere pentru scriere (*'w'* sau *'a'*) înseamnă că vor fi suportate și operații de citire din fișier. Dacă apare caracter *'+'* la un mod de deschidere pentru citire (*'r'*), atunci înseamnă că vor fi suportate și operații de scriere în fișier.

Iată câteva exemple de moduri de deschidere:

“rb” – Fișierul se deschide numai pentru citire în modul binar

“wb” – Fișierul se deschide numai pentru operații de scriere în modul binar

“ab” – Fișierul este deschis pentru adăugare la sfârșit în modul binar

“rb+” – Fișierul este deschis pentru citire în modul binar, dar sunt suportate și operații de scriere

“wb+” – Fișierul este deschis pentru scriere și citire în modul binar

“**ab+**” – fișierul este deschis pentru scriere cu adăugare la sfârșit în modul binar, dar se pot face și citiri din fișier.

Dacă mai sus nu apărea caracterul ‘*b*’ sau apărea caracterul ‘*t*’, atunci obțineam modurile similare text de deschidere de fișier.

Facem observația că semnul plus + poate fi pus și între cele două litere. De exemplu în loc de “**ab+**” putem pune “**a+b**”.

În exemplul următor deschidem un fișier pentru operații de citire, dar și cu posibilitate de scriere:

```
char numef[256];
FILE *fis;

printf("Numele fisierului: ");
gets(numef);
fis=fopen(numef, "rb+");
if (fis==NULL)
{
    printf("Eroare ! Nu am putut deschide fisierul %s.\n", numef);
    exit(1);
}
```

Închiderea fișierului se face cu ajutorul funcției *fclose*:

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

Dacă închiderea fișierului s-a făcut cu succes, atunci se returnează 0.

Pentru a închide toate fișierele deschise la un moment dat putem folosi funcția:

```
int fcloseall();
```

Funcția returnează numărul de fișiere ce au fost închise.

Verificarea dacă s-a ajuns la sfârșitul fișierului se face cu ajutorul funcției:

```
int feof(fis);
```

Funcția returnează 0 dacă nu s-a ajuns la sfârșitul fișierului sau o valoare nenulă dacă este sfârșit de fișier.

Există o funcție de golire a buffer-ului în fișier (forțează scrierea datelor din zona de memorie RAM tampon în fișier):

```
int fflush(fis);
```

În caz de succes se returnează 0, altfel se returnează valoarea corespunzătoare constantei *EOF* definită în *stdio.h*.

Golirea tuturor buffer-elor tuturor fișierelor deschise se face cu:

```
int flushall();
```

Verificarea dacă la un moment dat a apărut vreo eroare în prelucrarea unui fișier se face cu ajutorul funcției *ferror*, care returnează o valoare nenulă dacă există erori:

```
int ferror(fis);
```

Citirea unui caracter din fișier (și se returnează codul caracterului citit sau *EOF* dacă s-a ajuns la sfârșitul fișierului) se face cu:

```
int fgetc(fis);
```

Pentru a scrie un caracter în fișier folosim funcția:

```
int fputc(c, fis);
```

Citirea unui șir de caractere se face cu:

```
char *fgets(s, n, fis);
```

Pentru această funcție, fișierul este interpretat ca fiind text. Se citește un șir de caractere de lungime maximă n , șir care se depune la adresa s . Lungimea șirului depus în s este ori n (dacă nu s-a întâlnit caracterul ‘\n’ între primele n caractere ale liniei), ori numărul de caractere până la sfârșitul liniei. La sfârșitul șirului citit se depune caracterul ‘\0’.

Scrierea unui string într-un fișier text (la sfârșitul string-ului în fișier se pun caracterele cu codurile 10 și 13, adică este vorba de carriage-return ‘\r’ și respectiv new-line ‘\n’):

```
int fputs(s, fis);
```

Există o funcție pentru scriere formată într-un fișier text:

```
int fprintf(fis, ...);
```

Instrucțiunea este similară cu *printf* numai că în loc să se afișeze textul pe ecran, se depune în fișier.

Citire formatată în modul text (varianta *scanf* pentru fișiere text) este:

```
int fscanf (fis, ...);
```

Citire în modul binar se face cu:

```
size_t fread(void *p, int nb, int lb, FILE* fis);
```

Se citesc nb blocuri de lungime lb din fișierul indicat de pointerul fis și se depun la adresa de memorie p . Funcția returnează numărul efectiv de blocuri citite din fișier (*size_t* este un tip de date special definit pentru dimensiuni de blocuri de memorie).

Scrierea în modul binar se face cu:

```
size_t fwrite(void *p, int nb, int lb, fis);
```

Se iau nb blocuri de lungime lb de la adresa de memorie p și se scriu în fișierul fis . Funcția returnează numărul efectiv de blocuri scrise în fișier.

Există evident și o funcție pentru salt în fișier (mutarea poziției curente):

```
int fseek(FILE *fis, long n, int deunde);
```

Se face salt cu n octeți în fișierul *fis* din locul specificat de ultimul parametru al funcției. Parametrul *deunde* poate avea valorile 0, 1 sau 2, pentru fiecare dintre aceste valori fiind definite constante cu nume care sugerează poziția de pe care se face saltul:

SEEK_SET sau 0 - de la începutul fișierului
SEEK_CUR sau 1 - de la poziția curentă
SEEK_END sau 2 - de la sfârșitul fișierului.

Pentru *SEEK_SET*, n trebuie să fie nenegativ, deoarece saltul de la începutul fișierului se poate face numai spre dreapta.

Pentru *SEEK_END*, n trebuie să fie negativ sau 0, deoarece saltul se face înapoi, de la sfârșitul fișierului.

Pentru *SEEK_CUR*, n poate fi pozitiv, negativ sau 0. Saltul se face ori spre dreapta (dacă $n > 0$), ori înapoi de la poziția curentă (dacă $n < 0$).

Pentru returnarea poziției curente (în octeți de la începutul fișierului) vom folosi funcția:

```
long ftell(fis);
```

Folosind funcția *ftell* putem scrie o altă funcție care să returneze lungimea fișierului:

```
# include <stdio.h>

long flung(char *numef)
{
    FILE *fis;
    fis=fopen(numef,"rb");
    fseek(fis,0,SEEK_END);
    return ftell(fis);
}

void main(void)
{
    printf("Fișierul 'autoexec.bat' are %ld octeti.\n",
        flung("c:\\autoexec.bat"));
    getch();
}
```

Redenumirea unui fișier se face cu ajutorul funcției:

```
int rename(const char *numef1,const char *numef2);
```

Fișierul cu numele *numef1* se redenumeste cu numele *numef2*. Numele de fișiere din string-urile *numef1* și *numef2* pot fi precedate și de cale. Redenumirea nu se poate face decât pe același drive. De exemplu, apelul funcției *rename("c:\\fisier1.txt", "a:\\fisier2.txt")* se soldează cu eșec (nu se poate face redenumire de pe drive-ul *c*: pe drive-ul *a*:).

În cazul în care redenumirea s-a făcut cu succes, se returnează 0, altfel se returnează -1 și variabila globală *errno* se setează la una dintre valorile:

ENOENT – *No such file or directory* (nu există fișierul *numef1*, sau *numef2* este un nume de fișier invalid).
EACCES – *Permission denied* (nu există drepturi de scriere pe drive).
ENOTSAM – *Not same device* (situația din exemplul de mai sus, în care cele două nume de fișiere indică drive-uri diferite).

Ștergerea unui fișier de pe disc se face cu:

```
int remove(const char *numef);
```

Se încearcă ștergerea fișierului al cărui nume (eventual precedat de cale) este memorat în string-ul *numef*. În caz de succes, se returnează valoarea 0, altfel se returnează -1 și variabila globală *errno* se setează la una dintre valorile:

ENOENT – *No such file or directory* (nu există fișierul *numef* sau este invalid)
EACCES – *Permission denied* (nu există drepturi de scriere).

Dăm în final sursele a trei programe C care lucrează cu fișiere:

- 1) *Concatenarea mai multor fișiere ale căror nume sunt primite în linie de comandă. Ultimul parametru primit în linie de comandă se consideră a fi numele fișierului destinație:*

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

# define lbuf 1024 /* lungimea buffer-ului de transfer este 1KB */

int main(int narg, char *argv[])
{
    char *buf;
    int i, n, eroare=0;
    FILE *fis, *fisd;
    buf=(char*)malloc(lbuf);
    if (buf==NULL)
    {
        printf("Eroare! Memorie insuficienta.\n");
        getch(); return 1;
    }
    if (narg<3)
    {
        printf("Eroare! Structura comenzii este:\n\n");
        printf("%s <fis1> [<fis2>...<fisn>] <fis_dest>\n\n",
            argv[0]);
        printf("Se concateneaza <fis1>, <fis2> ... fisn.\n");
        printf("Fișierele se depun in <fis_destinatia>.\n");
        getch(); return 1;
    }
    fisd=fopen(argv[narg-1], "wb");
    if (fisd==NULL)
    {
        printf("Eroare! Nu am putut deschide %s pt. scriere.\n",
            argv[n-1]);
        getch(); return 0;
    }
    for (i=1; i<narg-1; i++)
```

```

{
    fis=fopen(argv[i],"rb");
    if (fis==NULL)
    {
        printf("Eroare! Nu am putut deschide %s.\n"
               ,argv[i]);
        eroare++;
    }
    do
    {
        n=fread(buf,1,lbuf,fis);
        fwrite(buf,1,n,fisd);
    }
    while (!feof(fis));
    fclose(fis);
}
fclose(fisd);
free(buf);
if (!eroare) printf("Concatenarea s-a incheiat cu succes !");
getch();
return 0;
}

```

2) Afişarea conţinutului unui fişier text:

```

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

#define maxlinie 79

int main(void)
{
    char numef[100],*linie;
    long n=0;
    FILE *fis;

    linie=(char*)malloc(maxlinie+1);
    if (linie==NULL)
    {
        printf("Eroare! Memorie insuficienta.\n");
        getch();
        return 1;
    }
    printf("Dati numele fisierului text de tiparit pe ecran: ");
    gets(numef);
    fis=fopen(numef,"rt");
    if (fis==NULL)
    {
        printf("Eroare! Nu am putut deschide %s pt. citire.\n"
               ,numef);
        getch();
        return 0;
    }
}

```

```

    }
    while(!feof(fis))
    {
        fgets(linie,maxlinie,fis);
        if (!feof(fis))
        {
            n++;
            printf(linie);
            if (n%24==0) getch();
        }
    }
    fclose(fis);
    free(linie);
    qprintf("Am afisat: %ld linii.",n);
    getch();
    return 0;
}

```

3) Produsul a două matrici citite din fișiere text.

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>

void EroareAloc()
{
    printf("Eroare! Memorie insuficienta.");
    exit(1);
}

float** MatrAlloc(int m,int n)
{
    int i;
    float **a;
    if ((a=(float**)calloc(m,sizeof(float*)))==NULL) EroareAloc();
    for (i=0;i<m;i++)
        if ((a[i]=(float*)calloc(n,sizeof(float)))==NULL)
            EroareAloc();
    return a;
}

void main(void)
{
    char numef[100];
    int i,j,k,m,n,n2,p;
    float **a,**b,**c,f;
    FILE *fis;
    printf("Dati numele fisierului text cu prima matrice: ");
    gets(numef);
    fis=fopen(numef,"rt");
    if (fis==NULL)

```

```

{
    printf("Eroare! Nu am putut deschide %s pt. citire.\n"
           ,numef);
    getch();
    exit(0);
}
fscanf(fis,"%d%d",&m,&n);
a=MatrAlloc(m,n);
for (i=0;i<m;i++)
{
    for (j=0;j<n;j++)
    {
        fscanf(fis,"%f",&f);
        a[i][j]=f;
        printf("%10.2lf",a[i][j]);
    }
    puts("");
}
fclose(fis);
printf("Dati numele fisierului text cu a doua matrice: ");
gets(numef);
fis=fopen(numef,"rt");
if (fis==NULL)
{
    printf("Eroare! Nu am putut deschide %s pt. citire.\n"
           ,numef);
    getch(); exit(0);
}
fscanf(fis,"%d%d",&n2,&p);
if (n!=n2)
{
    fclose(fis);
    printf("\nEroare! Matricile nu se pot inmulti:\n");
    printf(" A(%d,%d) x B(%d,%d)\n",m,n,n2,p);
    getch(); exit(0);
}
b=MatrAlloc(n,p);
for (i=0;i<n;i++)
{
    for (j=0;j<p;j++)
    {
        fscanf(fis,"%f",&f);
        b[i][j]=f;
        printf("%10.2lf",b[i][j]);
    }
    puts("");
}
fclose(fis);
printf("Dati numele fisierului in care sa pun rezultatul: ");
gets(numef);
if (fis==NULL)
{
    printf("Eroare! Nu am putut deschide %s pt. scriere.\n"

```

```

        ,numef);
    getch();
    exit(0);
}
fis=fopen(numef,"wt");
c=MatrAlloc(m,p);
for (i=0;i<m;i++)
    for (j=0;j<p;j++)
    {
        c[i][j]=0;
        for (k=0;k<n;k++) c[i][j]+=a[i][k]*b[k][j];
    }
for (i=0;i<m;i++)
{
    for (j=0;j<p;j++)
    {
        printf("%10.2f",c[i][j]);
        fprintf(fis,"%10.2f",c[i][j]);
    }
    printf("\r\n");
    fprintf(fis,"\r\n");
}
for (i=0;i<m;i++) free(a[i]);
for (i=0;i<n;i++) free(b[i]);
for (i=0;i<m;i++) free(c[i]);
free(a); free(b); free(c);
fclose(fis);
getch();
}

```

Pentru căutarea fișierelor pe disc în DOS ne folosim de funcțiile:

```

int findfirst(const char *cale, struct ffblk *ff, int atribut);
int findnext(struct ffblk *ff);

```

Funcțiile *findfirst* și *findnext* sunt definite în *dir.h* și *dos.h*.

Funcția *findfirst* inițiază căutarea, iar (dacă a fost găsit cel puțin un fișier) *findnext* o continuă. Cât timp se găsesc fișiere, *findnext* returnează valoarea 0.

În parametrul *cale* se transmite calea unde se face căutarea și structura numelor de fișier căutate. De exemplu, dacă parametrul *cale* conține "c:\\Windows*.bmp", atunci se vor căuta fișierele cu extensia *bmp* din directorul *windows* aflat pe drive-ul *c:*.

Variabila *ff* este un pointer către o structura *ffblk* în care se depun informațiile cu privire la fișierul găsit: numele său, dimensiunea (în octeți), data și ora creării, atributul fișierului (*Read-only*, *Hidden*, *Archive*, *System*). Structura *ffblk* este:

```

struct ffblk
{
    char ff_reserved[21]; /* rezervat */
    char ff_attrib;       /* atribut fisier */
    int ff_ftime;         /* ora creare fisier */
    int ff_fdate;         /* data creare fisier */
    long ff_fsize;        /* lungime fisier */
    char ff_name[13];     /* nume fisier */
}

```



```
};
```

Al treilea parametru al funcției *findfirst* reprezintă atributul fișierelor pentru care se inițiază căutarea. Acesta poate avea una dintre valorile:

FA_RDONLY - Read-only
FA_HIDDEN - Hidden
FA_ARCH - Archive
FA_SYSTEM - Sistem
FA_LABEL - Eticheta discului
FA_DIREC - Director

Ca exemplu prezentăm o funcție care afișează toate fișierele cu extensia *txt* din directorul curent:

```
void dirtxt(void)
{
    struct ffblk ff;
    int gata;
    puts("Fișierele *.txt din directorul curent:");
    gata = findfirst("*.txt",&ff,0);
    while (!gata)
    {
        printf("%16s",ff.ff_name);
        gata=findnext(&ff);
    }
}
```

Rezumat

Pentru a lucra cu un fișier în C trebuie să definim o variabilă de tip pointer către structura **FILE**. Deschiderea fișierului se face apelând funcția **fopen**. Funcția **fopen** primește ca parametri numele fișierului și modul în care vrem să deschidem fișierul. Dacă fișierul este deschis cu succes. Funcția **fopen** returnează pointerul către tipul FILE cu ajutorul căruia se va prelucra fișierul. Un fișier deschis cu **fopen** poate fi închis folosind funcția **fclose**. Scrierea/citirea în/din fișier în modul text se poate face cu **fprint**, respectiv cu **fscanf**. Pentru modul binar folosim funcțiile **fwrite** și **fread**. Poziționarea în fișier se face cu **fseek**.

Teme

1. Scrieți o funcție care returnează numărul de apariții a unui cuvânt într-un fișier text. Funcția primește ca parametrii cuvântul și numele fișierului.
2. Dintr-un fișier text se citesc elementele unui șir de numere reale. Să se sorteze crescător șirul și să se depună după aceea vectorul în același fișier.
3. Propunem să se scrie un program care să gestioneze stocul unei firme (vânzare și cumpărare cantitate dintr-un produs, adăugarea unui produs nou, afișarea unui produs după cod, căutarea unui produs după nume, calcularea valorii totale a stocului etc.). Un produs se va reține în fișier într-o înregistrare cu următoarea structură:

```
struct tstoc
{
```

```

    char cod_prod[10], den_prod[50];
    double cant, pret;
};

```

4. Să se caute și să se afișeze pe ecran toate numele fișierelor text și căile la care se găsesc pe drive-ul C: al calculatorului (căutarea se va face în toate directoarele drive-ului).

15. Fluxuri standard în C

Obiective

Ne propunem să vedem și alte fluxuri de date pe care putem aplica funcțiile prezentate la capitolul anterior.

În C (sub DOS) există definite următoarele fluxuri standard (vezi fișierul antet *stdio.h*):

stdin	- fluxul standard de intrare - tastatura
stdout	- fluxul standard de ieșire - monitorul
stderr	- fluxul standard de erori
stdaux	- interfața serială (COM1)
stdprn	- imprimanta (LPT1)

Fluxurile standard nu trebuie deschise sau închise, toate sunt în permanență deschise. Funcțiile pe care le-am prezentat pentru a lucra cu fișiere funcționează și pentru fluxurile standard (vezi capitolul anterior).

Ca aplicație prezentăm un program care tipărește la imprimantă un fișier text primit ca parametru în linie de comandă.

```

#include <stdio.h>

int main(int narg, char *argv[])
{
    FILE *fis;
    char c;
    if (narg!=2)
    {
        fprintf(stderr, "Eroare ! Utilizare incorecta.\n");
        return 1;
    }
    fis=fopen(argv[1], "rb");
    if (fis==NULL)
    {
        fprintf(stderr, "Eroare deschidere fisier.\n");
        return 1;    // fluxul standard de erori
    }
    while (fread(&c, 1, 1, fis) != NULL)    // citire caracter din fisier
        fprintf(stdout, "%c", c);    // trimitere la imprimanta
    fprintf(stdout, "%c", 12);    // trimitere caracter 12 la imprimanta
    fclose(fis);
    return 0;
}

```

```
}
```

Declanșarea efectivă a tipăririi la imprimantă se face trimițând caracterul cu codul 12. Dacă nu trimitem acest caracter, datele transmise rămân în principiu încărcate în buffer-ul imprimantei și se vor fi tipărit în momentul în care imprimanta va primi caracterul de declanșare a tipăririi, cel mai probabil când se va încerca o nouă tipărire.

Rezumat

Fluxurile standard C de sub DOS sunt: **stdin** (fluxul standard de intrare – tastatura), **stdout** (fluxul standard de ieșire – monitorul), **stderr** (fluxul standard de erori), **stdaux** (interfața serială – COM1) și **stdprn** (portul paralel – LPT1). Funcțiile prezentate la capitolul anterior pentru fișiere pot fi aplicate și acestor fluxuri.

Temă

Să se scrie o aplicație care trimite și primește pe portul serial sau paralel texte introduse de la tastatură. Să se instaleze și să se testeze această aplicație pe două calculatoare conectate prin cablu serial sau paralel (așa numita Direct Cable Connection).

16. Variabile statice

Obiective

Ne propunem să vedem ce sunt variabilele statice, cum se definesc și cum se utilizează.

În C, o variabilă locală într-o funcție poate fi declarată ca fiind statică. Pentru această variabilă se păstrează valoarea și, când se reapelează funcția, variabila va fi inițializată cu valoarea pe care a avut-o la apelul anterior. De fapt pentru variabila locală statică este alocată o zonă de memorie de la începutul până la sfârșitul execuției programului, dar accesul la variabilă nu este posibil decât din interiorul funcției în care este declarată. Iată un exemplu simplu:

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

```
void fct(void)
```

```
{
```

```
    static int x=1;
```

```
    x++;
```

```
    printf("%d ",x);
```

```
}
```

```
void main(void)
```

```
{
```

```
    int i;
```

```
    for (i=0;i<10;i++) fct();
```

```
}
```

Variabila statică *x* este inițializată la începutul execuției programului cu valoarea 1. La fiecare apel al funcției *fct* variabila *x* se incrementează cu o unitate și se afișează noua sa valoare. Așadar pe ecran în urma execuției programului va apărea:

2 3 4 5 6 7 8 9 10 11

Rezumat

O variabilă locală într-o funcție poate fi declarată ca fiind statică. O variabilă statică își păstrează valoare și după părăsirea funcției.

Temă

Folosiți variabile statice într-o funcție în care să rețineți data și ora ultimei apelări a funcției respective.

17. Funcții cu listă variabilă de argumente

Obiective

Ne propunem să studiem unul dintre cele mai interesante capitole ale limbajului C. Este vorba de posibilitatea de a scrie funcții ce pot fi apelate cu număr diferit de parametri.

În C putem defini funcții în care numărul de argumente (parametri) nu este fix. Putem apela o astfel de funcție cu oricâți parametri. Din această categorie fac parte funcțiile *printf* și *scanf*.

Ne propunem să scriem o funcție care să poată găsi maximul oricâtor valori transmise ca parametri, fără a folosi vectori.

O funcție cu listă variabilă de argumente se definește astfel:

```
tip_ret fctvarlist(tip prim, ...)
```

După numele funcției, între paranteze se dau tipul și denumirea primului parametru de apel al funcției, urmate de virgulă și de trei puncte. În consecință o funcție cu listă variabilă de argumente trebuie să aibă cel puțin un parametru când este apelată. Primul parametru de apel dă de obicei informații cu privire numărul de parametri de apel și eventual cu privire la tipul parametrilor, așa cum se întâmplă și în cazul funcțiilor *printf* și *scanf*.

Implementarea unei funcții cu listă variabilă de argumente se face cu ajutorul macro-urilor *va_start*, *va_arg* și *va_end*. Aceste macro-uri sunt definite în fișierul antet *stdarg.h*.

Pentru a identifica parametrii de apel, în interiorul corpului unei funcții având listă variabilă de argumente se definește o variabilă de tip *va_list*, care este un vector definit în fișierul antet *stdarg.h*. Acest vector reține practic informațiile cu privire la transmiterea parametrilor. Pe baza acestor informații și cu ajutorul celor trei macro-uri (*va_start*, *va_arg* și *va_end*) se pot identifica valorile parametrilor de apel ai funcției.

Macro-ul *va_start* setează vectorul *va_list* așa încât să indice primul parametru de apel al funcției. El are doi parametri. Primul este vectorul de tip *va_list*, iar al doilea este variabila în care se depune valoarea primului parametru de apel al funcției. Această variabilă trebuie să fie dată explicit ca prim argument în definiția funcției.

Apelând succesiv macro-ul *va_arg*, se identifică restul parametrilor de apel ai funcției (de la al doilea până la ultimul). Macro-ul *va_arg* se transformă într-o expresie (vezi capitolul dedicat macro-urilor) care are același tip și aceeași valoare cu următorul parametru de apel al funcției. Expresia se generează pe baza celor doi parametri ai macro-ului. Primul parametru este vectorul de tip *va_list*, iar al doilea este tipul la care se convertește expresia. Acest tip de dată trebuie evident să fie cel al parametrului de apel.

Trebuie reținut faptul că datorită mecanismului promovării din C (convertirea automată a unui tip la unul superior din aceeași categorie) funcțiile cu listă variabilă de parametri nu funcționează corect pentru tipurile de dată *char*, *unsigned char* și *float*.

Macro-ul *va_end* se apelează după ce toți parametrii de apel ai funcției cu listă variabilă de argumente au fost identificați. Acest macro ajută ca ieșirea din funcție să se facă în condiții bune.

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

double max(int n,...) /*functie cu lista variabila
                        de argumente */
{
    va_list lista_arg;
    int i;
    double max=-1E10, arg;

    va_start(lista_arg, n);
    for (i=0; i<n; i++)
    {
        arg=va_arg(lista_arg, double);
        if (arg>max) max=arg;
    }
    va_end(lista_arg);
    return max;
}

void main()
{
    printf("Maximul este: %lf\n", max(5, -1., 5., -3.7, 4.9, 0.));
    getch();
}
```

Să menționăm faptul că dacă nu se inițializează lista de argumente (vectorul de tip *va_list*) cu un apel al macro-ului *va_start*, dacă se apelează *va_arg* de mai multe ori decât numărul parametrilor de apel sau dacă se omite *va_end*, programul se poate bloca sau rezultatele pot fi neașteptate. De asemenea, trebuie să avem mare grijă ca tipul parametrilor de apel să coincidă cu tipul de date transmis ca parametru macro-ului *va_arg*. În exemplul nostru, dacă nu transmitem toți cei cinci ultimi parametri de tip *double*, puteam obține rezultatul eronat.

Marele dezavantaj al funcțiilor cu listă variabilă de argumente este dat de faptul că nu putem controla din corpul funcției numărul și tipul parametrilor de apel.

În final trebuie să facem observația că limbajul C++ oferă câteva alternative mai simple de implementat și mai sigure la funcțiile cu listă variabilă de argumente.

Rezumat

În C este posibil să definim funcții ce pot fi apelate cu număr variabil de parametri, așa cum este cazul funcțiilor **printf** sau **scanf**. Aceste funcții se implementează cu ajutorul tipului **va_list** și a macrocomenzilor **va_start** și **va_end**, toate definite în fișierul antet **stdarg.h**.

Teme

1. Să se scrie o funcție cu număr variabil de argumente care calculează media aritmetică a unor valori reale.
2. Să se scrie o funcție cu număr variabil de argumente care calculează media aritmetică dintre maximul și minimul unor valori reale.

18. Utilizarea modului text de afișare al adaptorului video

Obiective

Ne propunem să vedem care sunt funcțiile Borland C scrise pentru utilizarea modului text sub MS-DOS.

Fiecărui caracter de pe ecran în modul text i se atașează în memoria video câte 2 octeți. Unul dintre octeți conține codul caracterului, iar al doilea conține informații cu privire la modul de afișare al acestuia. Astfel primii 4 biți reprezintă culoarea caracterului. Cei 4 biți conduc la valori întregi în baza 10 între 0 și 15, adică valori corespunzătoare celor 16 culori. Următorii 3 biți reprezintă culoarea fundalului pe care se scrie caracterul, iar ultimul bit este rezervat pentru clipire: 1 - cu clipire și 0 - fără.

Pentru schimbarea culorii caracterelor avem instrucțiunea:

```
void textcolor(int c);
```

c este o valoare întreagă între 0 și 15, reprezentând una dintre cele 16 culori ale modului text. Pentru fiecare culoare este definită câte o constantă:

Denumire constantă pentru culoare	Valoare constantă
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
DARKGRAY	7
LIGHTGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14

WHITE	15
-------	----

Din cele 16 culori de mai sus numai primele 8 (nuanțele închise de la 0 la 7) pot fi folosite pentru setarea culorii fundalului pe care se afișează textul. Schimbarea culorii fundalului se face cu funcția:

```
void textbackground(int c)
```

Toate atributele caracterelor pot fi schimbate simultan cu ajutorul instrucțiunii:

```
void textattr(int attr);
```

Parametrul *attr* poate lua o valoare între 0 și 255, reprezentând toate combinațiile de culori pentru text, fundal și clipire. Dăm un exemplu:

```
textattr(BLINK, YELLOW, (RED<<4));
```

În exemplul de mai sus se setează culoarea caracterelor ca fiind galbenă, culoarea fundalului roșie, cu clipire.

Pentru clipire este definită constanta BLINK, care are valoarea 128.

Putem trece de la un mod text la altul cu ajutorul funcției:

```
void textmode(int mod);
```

Exista 5 moduri text. Fiecare mod text corespunde unei valori întregi:

Denumire constantă	Valoare constantă	Semnificație
BW40	0	Mod text 40 caractere pe linie cu 2 culori: alb/negru
C40	1	40 caractere pe linie cu 16 culori
BW80	2	80 caractere pe linie cu 2 culori
C80	3	80 caractere pe linie cu 16 culori
MONO	7	80 caractere pe linie, 16 nuanțe de gri
LAST MODE	-1	Se revine la ultimul mod text

Definirea ferestrelor text se face cu ajutorul funcției:

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

Se definește o fereastră text având colțul stânga-sus la coordonatele ecran (*x1*, *y1*) și colțul dreapta-jos (*x2*, *y2*). Coordonatele ecranului text sunt numerotate pe orizontală de la stânga spre dreapta (de la 1 la 40 sau 80, în funcție de modul text) și pe verticală de sus în jos (de la 1 la 25).

Funcțiile *printf* și *scanf* nu țin cont de fereastra text definită și de culorile setate. Când se dorește a se lucra cu ferestre text se utilizează funcțiile *cprintf* și *cscanf*.

Pentru returnarea poziției cursorului pe ecran, pe orizontală, respectiv pe verticală, se folosesc funcțiile:

```
int wherex();
int wherey();
```

Mutarea cursorul la coordonatele (x, y) de pe ecran se face cu funcția:

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

Afișarea cu intensitate sporită a caracterelor se face cu:

```
void highvideo();
```

Afișarea cu intensitate redusă a caracterelor se face cu funcția:

```
void lowvideo();
```

Revenirea la modul text normal se face cu:

```
void normvideo();
```

Caracteristicile ferestrei text curente se pot obține cu ajutorul funcției:

```
void gettextinfo(struct text_info *info);
```

Structura *text_info* este:

```
struct text_info
{
    unsigned char winleft;          /* coordinata stanga a ferestrei */unsigned char
    wintop;          /* coordinata sus a ferestrei */unsigned char winright;          /*
    coordinata dreapta a ferestrei */unsigned char winbottom;          /* coordinata jos a
    ferestrei */
    unsigned char attribute;        /* atributul text */
    unsigned char normattr;        /* atributul normal */
    unsigned char curmode;          /* mod text curent:
                                   BW40, BW80, C40, C80, sau C4350 */
    unsigned char screenheight;     /* inaltimea ecranului */
    unsigned char screenwidth;      /* latimea ecranului */
    unsigned char curx;             /* coordonata x a cursorului */
    unsigned char cury;            /* coordonata y a cursorului */
};
```

Alte funcții ale modului text (definite în *conio.h*):

1. *void clrscr()*; curăță fereastra text curentă.
2. *void clreol()*; curăță linia curentă începând cu caracterul curent;
3. *void insline()*; inserează linie nouă goală pe poziția în care se află cursorul, liniile începând cu linia curentă coborând cu o poziție în jos;
4. *void movetext(int x1,int y1,int x2,int y2,int x3,int y3)*; mută textul aflat pe ecran în interiorul dreptunghiului de coordonate stânga-sus (*x1, y1*) și dreapta-jos (*x2, y2*) la coordonatele stânga-sus (*x3, y3*).
5. *int gettext(int x1,int y1,int x2,int y2,void *buff)*; citește caracterele aflate în interiorul dreptunghiului stânga-sus (*x1, y1*) și dreapta-jos (*x2, y2*) și le depune la adresa de memorie indicată de *buff*. Zona de memorie alocată la adresa *buff* trebuie să fie de minim: $2*(x2-x1+1)*(y2-y1+1)$ octeți. În cazul în care citirea s-a făcut cu succes, funcția returnează valoarea 1, altfel returnează valoarea 0.
6. *int puttext(int x1,int y1,int x2,int y2,void *buff)*; de la adresa indicată de *buff* se iau $2*(x2-x1+1)*(y2-y1+1)$ octeți și se afișează în dreptunghiul de pe ecran dat de

coordonatele stânga-sus ($x1, y1$) și dreapta-jos ($x2, y2$). În cazul în care afișarea s-a făcut cu succes, funcția returnează o valoare nenulă.

Observație: Cele 6 funcții de mai sus țin cont de fereastra text curent definită, coordonatele fiind relativ la aceasta.

Rezumat

Curățarea ecranului în modul text se face apelând funcția **clrscr()**, schimbarea culorii caracterelor ce se afișează se face cu ajutorul funcției **textcolor**, schimbarea culorii fundalului se face cu **textbackground**. Definirea unei ferestre text se face cu funcția **window**. Funcțiile de afișare, respectiv de citire care țin cont de fereastra text definită și de culorile curente stabilite sunt **cprintf** și **cscanf**.

Temă

Scrieți o funcție care afișează și pregătește scrierea în interiorul unei ferestre text având un contur format cu o linie dublă. Fereastra trebuie să semăne cu ferestrele mediului de programare Borland C de sub DOS.

19. Grafică în C

Obiective

Ne propunem să studiem funcțiile de lucru în modul grafic din Borland C pentru sistemul de operare MS-DOS.

În C implicit se lucrează în modul text. Pentru reprezentări grafice trebuie să intrăm în modul grafic.

Dacă folosim instrucțiuni ale modului grafic în modul text se semnalează eroare ce duce la părăsirea programului cu mesajul de eroare: *BGI Error: Graphics not initialized (use 'initgraph')*. Instrucțiuni din modul text pot fi însă folosite în modul grafic, fără a se semnala eroare.

Tot ceea ce ține de modul grafic (funcții, constante, variabile globale etc.) este definit în fișierul antet *graphics.h*, care trebuie inclus în programele care folosesc grafică. Funcțiile modului grafic sunt implementate în biblioteca *graphics.lib*.

În modul grafic ecranul este interpretat ca o rețea dreptunghiulară de puncte (pixeli). Punctele de pe ecran sunt numerotate de la stânga spre dreapta pe orizontală, iar pe verticală de sus în jos. Coordonatele punctelor de pe ecran sunt numerotate începând cu valoarea 0. De exemplu, în modul VGA 640x480, pixelii de pe ecran au coordonatele (x,y), unde $x \in \{0, 1, 2, \dots, 639\}$ și $y \in \{0, 1, 2, \dots, 479\}$

19.1. Inițializarea modului grafic

Pentru a putea folosi instrucțiunile de desenare trebuie inițializat modul grafic. Acest lucru se face cu funcția:

```
initgraph(&gd, &gm, "c:\");
```

gd reprezintă valoarea corespunzătoare plăcii grafice (*graphic device*).

gm reprezintă modul grafic în care dorim să intrăm (*graphic mode*).

cale este o constantă tip șir de caractere corespunzătoare căii spre fișierele necesare inițializării modului grafic.

Aceste fișiere au extensia *bgi*. Ele se găsesc de obicei în directorul *bgi* aflat în directorul în care este instalat mediul de programare *C* (pentru *Borland C* avem de obicei calea *c:\bc\bgi*, deoarece mediul *Borland C* se instalează în mod implicit în directorul *bc*, în rădăcina harddiscului *c:*). Dacă lăsăm calea vidă (stringul vid ""), atunci fișierele grafice sunt căutate în directorul curent). Este indicat chiar să se lase vidă calea spre *bgi* și să se copieze fișierele grafice necesare în directorul curent, așa nedepinzându-se mediul de programare, programul împreună cu fișierele grafice putând fi copiate pe orice calculator.

Mai mult, fișierele grafice pot fi incluse direct în executabil prin transformarea acestora în fișiere *obj* cu ajutorul utilitarului *bgiobj.exe*, aflat în directorul *bgi*.

În momentul inițializării modului grafic, dacă variabila *gd* are valoarea corespunzătoare constantei *DETECT* (adică 0), atunci este căutat modul grafic cel mai bun în care se poate intra. În prezent, datorită faptului că aproape toate calculatoarele au placă video Super VGA, implicit modul grafic găsit va fi cel VGAHI (16 culori, rezoluție 640x480). Pentru a intra în acest mod este nevoie ca în calea în care se caută fișierele grafice să se afle fișierul *egavga.bgi*.

Exemplu. Inițializare grafică prin detectarea modului grafic.

```
int gd=DETECT, gm;  
initgraph(&gd, &gm, "c:\\bc\\bgi");
```

Parametrii *gd* și *gm* sunt folosiți și ca parametri de intrare în funcție și ca parametri de ieșire din funcție. De aceea, în exemplul de mai sus *gd* intră cu valoarea 0, corespunzătoare constantei *DETECT* și se returnează în *gd* valoarea corespunzătoare constantei *VGA*, adică 9, iar pentru *gm* se returnează valoarea corespunzătoare constantei *VGAHI*, adică 2.

Dacă dorim să se inițializeze un anumit mod grafic, corespunzător unei anumite plăci grafice, nu avem decât să specificăm acest lucru prin inițializarea corespunzătoare a variabilelor *gd* și *gm*.

Valorile pe care le poate lua *gd* sunt:

Denumire constantă placă video	Valoare constantă
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Valorile posibile pentru *gm* (depinzând de placa video aleasă) sunt:

Constantă placă video	Constantă pentru mod grafic	Valoare constantă	Rezoluție	Paleta de culori	Nr. pag. grafice
CGA	CGAC0	0	320x200	C0	1

	CGAC1	1	320x200	C1	1
	CGAC2	2	320x200	C2	1
	CGAC3	3	320x200	C3	1
	CGAHI	4	640x200	2 culori	1
MCGA	MCGAC0	0	320x200	C0	1
	MCGAC1	1	320x200	C1	1
	MCGAC2	2	320x200	C2	1
	MCGAC3	3	320x200	C3	1
	MCGAMED	4	640x200	2 culori	1
	MCGAHI	5	640x480	2 culori	1
EGA	EGALO	0	640x200	16 culori	4
	EGAHI	1	640x350	16 culori	2
EGA64	EGA64LO	0	640x200	16 culori	1
	EGA64HI	1	640x350	4 culori	1
EGAMONO	EGAMONOH	3	640x350	2 culori	2
HERC	HERCMONOH	0	720x348	2culori	2
ATT400	ATT400C0	0	320x200	C0	1
	ATT400C1	1	320x200	C1	1
	ATT400C2	2	320x200	C2	1
	ATT400C3	3	320x200	C3	1
	ATT400MED	4	640x200	2 culori	1
	ATT400HI	5	640x400	2 culori	1
VGA	VGALO	0	640x200	16 culori	2
	VGAMED	1	640x350	16 culori	2
	VGAHI	2	640x480	16 culori	1
PC3270	PC3270HI	0	720x350	2 culori	1
IBM8514	IBM8514LO	0	640x480	256 culori	1
	IBM8514HI	1	1024x760	256 culori	1

Exemplu. Inițializare mod grafic VGAMED (16 culori, rezoluție 640x350 și 2 pagini grafice).

```
int gd=VGA, gm=VGAMED;
initgraph(&gd, &gm, "c:\\bc\\bgi");
```

Observație: 1) Nu se poate intra în oricare dintre modurile grafice de mai sus de pe orice calculator! Trebuie să se țină cont de compatibilitatea dintre placa existentă și cea pentru care se încearcă inițializarea grafică. Pe orice calculator aparținând ultimelor generații funcționează în general inițializarea grafică EGA sau VGA.

2) Despre pagini grafice vom vorbi când se vor prezenta funcțiile *setvisualpage* și *setactivepage*.

Dacă vrem să inițializăm un alt mod grafic decât cele cunoscute, de exemplu un mod *Super VGA* cu 256 culori (pentru care trebuie să avem fișierul *svga256.bgi*), înainte de inițializare trebuie făcută o instalare (cu DETECT nu sunt găsite modurile Super VGA). În exemplul următor este arătat modul în care este instalat driver-ul pentru placa Super VGA cu 256 culori.

```
void igSVGA256(int gm)
{
    int eroare_grafica, gd;
    gd = installuserdriver("SVGA256", NULL);
```

```

initgraph(&gd,&gm,"c:\\bc\\bgi");
eroare_grafica = graphresult();
if (eroare_grafica != grOk)
{
    printf("Graphics error: %s\n", grapherrormsg(eroare_grafica));
    getch();
    exit(1);
}
}

```

Observație: 1) Pentru modurile grafice Super VGA cu 256 culori nu există definite constante, deoarece ele nu au existat când a apărut **Borland C**. Modurile grafice (trimise ca parametru în funcția *igSvGA256* sunt numerotate cu 0, 1, 2, 3, 4,, aceste valori corespunzând rezoluțiilor: 320x200, 640x400, 640x480, 800x600, 1024x768 etc.).

2) La modurile Super VGA cu 256 culori există o singură pagină grafică.

3) În anumite sisteme de operare (de exemplu **Windows NT**) inițializarea modurilor Super VGA nu este posibilă. În general nu există probleme sub DOS, Windows '95 sau Windows '98.

4) Există drivere scrise numai pentru anumite plăci video, cum ar fi *Vesa16.bgi* sau *S3trio64.bgi*, care de asemenea trebuie instalate pentru a putea fi folosite. Aceste moduri grafice nu pot fi inițializate decât pentru plăcile video respective, pentru alte plăci nefuncționând.

2. getaspectratio(xx,yy);

În unele moduri grafice (cum ar fi VGAMED si VGALO) apare o alungire a imaginii pe verticală, deoarece pixelul este mai mare pe verticală decât pe orizontală. De aceea un cerc arată ca o elipsă, un pătrat ca un dreptunghi etc. Raportul $def = (float)xx/yy$ ne indică de câte ori e mai mare pixelul pe verticală decât pe orizontală. Pentru a desena păstrând raportul între dimensiunile figurii, ordonata la fiecare desen va fi înmulțită cu raportul de deformare *def*.

Exemplu. afișare corectă a unui disc.

```

int gd=VGA,gm=VGAMED,xx,yy,x=200,y=100,rx=50,ry; // rx=raza discului
float def;
/* ..... */
getaspectratio(xx,yy);
def=(float)xx/(float)yy; // calculare raport deformare
ry=rx*def; // rectificare raza pe verticala
fillellipse(x,y,rx,ry); // desena disc
/* ..... */

```

3. Funcția graphresult() returnează o valoare ce indică modul în care s-a terminat inițializarea modului grafic. Dacă valoarea coincide cu valoarea constantei *grOk*, atunci inițializarea s-a făcut cu succes, iar dacă nu, atunci folosind valoarea returnată de funcția *graphresult* putem afișa mesajul corespunzător valorii furnizate de tentativa de inițializare a modului grafic.

4. Mesajul de eroare generat este dat de funcția: **grapherrormsg(val)**, unde *val* este indicele erorii pentru care se afișează mesajul.

5. Când terminăm de lucrat în modul grafic și vrem să ne întoarcem în modul text folosim instrucțiunea de închidere a modului grafic: **closegraph();**

Exemplu. Schema unui program în modul grafic, inițializare si închidere mod grafic:

```

#include<graphics.h>
#include<process.h>
#include<stdlib.h>

void main(void)
{
    int gd=DETECT,gm,eroare;
    initgraph(&gd,&gm,"c:\\bc\\bgi");
    err=graphresult();
    if (eroare!=gok)
    {
        printf("Eroare grafica %d: %s",eroare,grapherrormsg(eroare));
        exit(1);
    }

    ...
    closegraph();
}

```

6. getmaxx() și getmaxy() returnează indicele pe orizontala (coloana) ultimului pixel de pe ecran, respectiv linia ultimului pixel de pe ecran. Pixelii pe orizontala ecranului sunt numerotați de la stânga spre dreapta, de la **0** la **getmaxx()**, iar pe verticală sunt numerotați de la sus în jos, de la **0** la **getmaxy()**.

De exemplu, în modul VGAHI (640x480), **getmaxx()** va returna 639, iar **getmaxy()** va returna 479.

7. getmaxcolor() returnează indicele ultimei culori a paletei. Pentru oricare dintre modurile VGA, valoare returnată va fi 15 (culorile vor avea indici de la 0 la 15). Pentru fiecare din cele 16 culori ale paletei avem definită câte o constantă. Putem folosi aceleași constante ca la modul text:

Denumire constantă pentru culoare	Valoare constantă
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
DARKGRAY	7
LIGHTGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

Valorile pentru culorile paletei VGA și EGA nu sunt însă consecutive, de aceea, dacă schimbăm paleta de culori, trebuie să ținem cont de acest lucru. Cele 16 culori ale paletei

corespund valorilor 1, 2, 3, 4, 5, 7, 20, 56, 57, 58, 59, 60, 61, 62 și 63. Pentru fiecare dintre aceste culori este definită câte o constantă:

Denumire constantă	Valoare constantă
EGA_BLACK	0
EGA_BLUE	1
EGA_GREEN	2
EGA_CYAN	3
EGA_RED	4
EGA_MAGENTA	5
EGA_LIGHTGRAY	7
EGA_BROWN	20
EGA_DARKGRAY	56
EGA_LIGHTBLUE	57
EGA_LIGHTGREEN	58
EGA_LIGHTCYAN	59
EGA_LIGHTRED	60
EGA_LIGHTMAGENTA	61
EGA_YELLOW	62
EGA_WHITE	63

Exemplu. Afișarea caracteristicilor modului grafic detectat.

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

int main(void)
{
    int gd=DETECT,gm,eroare;
    char s[80];
    initgraph(&gd,&gm,"c:\\bc\\bgi");
    err=graphresult();
    if (eroare!=grok)
    {
        printf("Eroare grafica %d: %s",eroare,grapherrormsg(eroare));
        return 1;
    }
    sprintf(s,"Placa grafica %d, mod grafic %d, %d culori, rez. %d x %d",
        gd,gm,getmaxcolor()+1,getmaxx()+1,getmaxy()+1);
    outtext(s); /* afisare string s la coordonatele grafice curente */
    getch(); closegraph(); return 0;
}
```

8. cleardevice(); curată ecranul în modul grafic.

9. setbkcolor(c); setează culoarea fundalului pentru întreg ecranul, unde c este constantă întreagă. Culoarea fundalului nu se schimbă decât după un apel al funcției **cleardevice()**.

10. getbkcolor() funcție ce returnează culoarea curentă a fundalului.

11. moveto(x,y); mută cursorul grafic la coordonatele grafice x și y.

12. moverel(dx,dy); mută cursorul grafic la coordonatele dx și dy , relative la fostele coordonate curente. Dacă avem coordonatele curente $x0$ și $y0$, atunci în urma apelului **moverel(dx,dy);** coordonatele curente devin $x0+dx$ și $y0+dy$.

13. getx() și **gety()** returnează coordonatele grafice curente pe orizontală, respectiv pe verticală.

14. putpixel(x,y,c); afișează un punct (aprinde un pixel) pe ecran la coordonatele (x,y) având culoarea c .

Exemplu. Umplerea ecranului cu pixeli de culori aleatoare.

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

void main(void)
{
    int i,j,maxx,maxy,gd=DETECT,gm;
    initgraph(&gd,&gm,"");
    randomize();
    maxx=getmaxx(); maxy=getmaxy();
    for (i=0;i<=maxx;i++)
        for (j=0;j<=maxy;j++) putpixel(i,j,random(16));
    getch(); closegraph();
}
```

Observatie: Funcția *random(n)* definită în fisierul antet *stdlib.h* primește ca argument un întreg n și returnează o valoare întreagă aleatoare între 0 și $n-1$. Funcția *randomize()* inițializează generatorul aleator de numere cu o valoare aleatoare depinzând de timp. Dacă nu se apelează funcția *randomize*, atunci de fiecare data când se rulează programul, se generează aceleași numere aleatoare.

15. getpixel(x,y) citește culoarea pixelului aflat la coordonatele (x,y) și returnează această valoare (o valoare între 0 și *getmaxcolor()*).

Caracteristicile contururilor

16. setcolor(c); setează culoarea cu care se vor desena contururile desenelor (linii, curbe etc.).

17. getcolor(); returnează indicele culorii curente de desenare a contururilor (o valoare între 0 și *getmaxcolor()*).

18. setlinestyle(stil, model_utilizator, grosime); setează tipului de linie cu care se vor desena contururile.

stil dă tipul liniei (contururilor), adică unul dintre cele definite în tabelul:

Constantă pentru tip linie	Valoare constantă	Descriere
SOLID_LINE	0	linie continuă
DOTTED_LINE	1	linie punctată
CENTER_LINE	2	linie – punct

DASHED_LINE	3	linie întreruptă
USERBIT_LINE	4	definită de utilizator

model_utilizator este o valoare întreagă (pe 2 octeți) folosită pentru definirea de către programator a unui tip de linie propriu.

grosime setează grosimea liniilor contururilor. Acest parametru poate lua două valori, 1 sau 3:

Constantă pentru tip linie	Valoare constantă	Descriere
NORM_WIDTH	1	linie subțire (grosime 1 pixel)
THICK_WIDTH	3	linie subțire (grosime 3 pixel)

Ne propunem să definim următorul tip de linie: 00000000x00xxx00, unde prin “o” înțelegem spațiu, iar prin “x” înțelegem punct. Fiecare spațiu corespunde valorii 0, iar “x” valorii 1 (în baza 2). Modelul de mai sus corespunde, deci, numărului $0000000010011100_{(2)} = 2^7 + 2^4 + 2^3 + 2^2_{(10)} = 156_{(10)} = 009C_{(16)}$. Pentru a defini și folosi acest model vom scrie:

```
int model=156;
setlinestyle(USERBIT_LINE, model, NORM_WIDTH);
```

19.2. Instrucțiuni pentru desenare de linii drepte

19. line(x1,y1,x2,y2); desenează un segment de dreaptă între punctele de coordonate $(x1,y1)$ și $(x2,y2)$. Coordonatele capătului final al segmentului devin coordonatele grafice curente, adică în punctul de coordonate $(x2,y2)$.

20. lineto(x,y); trasează un segment de dreaptă pornind din punctul de coordonate grafice curente până în punctul de coordonate (x,y) . Coordonatele grafice curente se mută în capătul final al segmentului, adică în punctul de coordonate (x,y) .

21. linerel(x,y); trasează un segment de dreaptă pornind din punctul de coordonate grafice curente până în punctul obținut printr-o deplasare (dx,dy) față de punctul curent. Coordonatele capătului final al segmentului devin coordonatele grafice curente, adică punctul de coordonate $(x+dx,y+dy)$.

22. rectangle(x1,y1,x2,y2); se desenează conturul unui dreptunghi având 2 vârfuri diametral opuse de coordonate $(x1,y1)$ și $(x2,y2)$.

Exemplu. Trasarea unui pătrat folosind funcțiile: *line*, *lineto*, *linerel*, *rectangle*.

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

void main(void)
{
    int x=200,y=100,l=70,gd=DETECT,gm;
    initgraph(&gd,&gm,"");
    setbkcolor(BLUE);
    cleardevice();
    setlinestyle(SOLID_LINE,0,1);
```



```

setcolor(WHITE);
line(x,y,x+L,y);
line(x+L,y,x+L,y+L);
line(x+L,y+L,x,y+L);
line(x,y+L,x,y);
getch();
setcolor(YELLOW);
setlinestyle(DOTTED_LINE,0,3);
moveto(x,y);
lineto(x+L,y);
lineto(x+L,y+L);
lineto(x,y+L);
lineto(x,y);
getch();
setcolor(LIGHTCYAN);
setlinestyle(CENTER_LINE,0,1);
moveto(x,y);
linereel(L,0);
linereel(0,L);
linereel(-L,0);
linereel(0,-L);
getch();
setcolor(LIGHTGREEN);
setlinestyle(DASHED_LINE,0,3);
rectangle(x,y,x+L,y+L);
getch();
closegraph();
}

```

23. drawpoly(n,p); trasează o linie poligonală, unde n este numărul de vârfuri al liniei poligonale, iar în p sunt memorate coordonatele curente ale vârfurilor poligonului.

În vectorul p sunt memorate $2*n$ valori întregi (coordoanatele celor n vârfuri). Pentru vârful de indice k avem coordonatele $(p[2*k], p[2*k+1])$, unde $k \in [0, n-1]$.

Exemplu. Se citesc coordonatele vârfurilor unui poligon și se desenează.

```

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

void main(void)
{
    int k,n,p[100],gd=DETECT,gm;
    clrscr();
    printf(" Dati numarul de varfuri al poligonului: ");
    scanf("%d",&n);
    for (k=0;k<n;k++)
    {
        printf("x(%d)= ",k);
        scanf("%d",&p[2*k]);
        printf("y(%d)= ",k);
        scanf("%d",&p[2*k+1]);
    }
}

```

```

p[2*n]=p[0]; p[2*n+1]=p[1];
n++;
initgraph(&gd,&gm,"");
drawpoly(n,p);
getch(); closegraph();
}

```

Observație: Pentru a închide linia poligonală, la sfârșitul vectorului p a fost necesară adăugarea coordonatelor primului vârf.

19.3. Instrucțiuni pentru desenare de curbe eliptice

24. circle(x,y,r); se desenează un cerc cu centrul în punctul de coordonate (x,y) și având raza r .

Exemplu. Trasarea unor cercuri concentrice.

```

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

void main(void)
{
    int x,y,r,n=20,gd=DETECT,gm;
    initgraph(&gd,&gm,"");
    x=getmaxx()/2;
    y=getmaxy()/2;
    for (r=0;r<n;r++) circle(x,y,5*r+50);
    getch();
    closegraph();
}

```

25. arc(x,y,u1,u2,r); se desenează un arc de cerc cu centrul în punctul de coordonate (x,y) și având raza r , între unghiurile $u1$ și $u2$ date în grade, în sens trigonometric.

26. ellipse(x,y,u1,u2,rx,ry); se trasează un arc de elipsă cu centrul în punctul de coordonate (x,y) , de raze rx și ry , între unghiurile $u1$ și $u2$ date în grade, în sens trigonometric. Pentru desenarea unei elipse întregi putem folosi funcția sub forma **ellipse(x,y,0,360,rx,ry);**.

19.4. Instrucțiuni pentru afișare de texte în modul grafic

27. outtext(s); afișează șirul de caractere memorat în variabila de tip string s , la coordonatele grafice curente.

28. outtextxy(x,y,s); textul din stringul s este afișat la coordonatele grafice (x,y) de pe ecran.

29. settextstyle(font,directie,dim); setează caracteristicile textelor ce vor fi afișate în modul grafic.

font reprezintă tipul caracterelor. Tipurile caracterelor (fonturile) sunt definite în fișiere cu extensia *chr* aflate în directorul *bgi*. În continuare dăm tabelul cu constantele pentru diferite fonturi:

Denumire constantă	Valoare constantă
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4

directie poate lua 2 valorile *HORIZ_DIR*, adică 0, respectiv *VERT_DIR*, adică 1. Acest parametru dă direcția de scriere (pe orizontală, sau pe verticală).

dim este o valoare întreagă reprezentând dimensiunea caracterelor. 1 este dimensiunea normală (8x8 pixeli), 2 este de două ori mai mare (16x16 pixeli), 3 de trei ori mai mare (24x24 pixeli) etc.

30. Setarea modului de afișare al textului (alinieri pe orizontală și verticală) se face cu ajutorul funcției **settextjustify(o,v);**

Modurile de aliniere pe orizontală sunt:

Denumire constantă	Valoare constantă	Descriere
LEFT_TEXT	0	Aliniere la stânga
CENTER_TEXT	1	Text centrat
RIGHT_TEXT	2	Aliniere la dreapta

Modurile de aliniere pe verticală:

Denumire constantă	Valoare constantă	Descriere
BOTTOM_TEXT	0	Aliniere la stânga
CENTER_TEXT	1	Text centrat
TOP_TEXT	2	Aliniere la dreapta

Exemplu. Să se deplaseze folosind butoanele direcționale (săgețile) un text pe ecranul monitorului.

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

```
void main(void)
{
    char c,s[50];
    int x,y,pas=5,gd=DETECT,gm;

    clrscr();
    printf("Dati un text: ");
    gets(s);
    initgraph(&gd,&gm,"");
    settextstyle(GOTHIC_FONT,0,2);
    settextjustify(CENTER_LINE,CENTER_LINE);
```

```

setcolor(YELLOW);
x=getmaxx()/2;
y=getmaxy()/2;
do
{
    outtextxy(x,y,s);
    if (kbhit())
    {
        c=getch();
        if (kbhit())
        {
            c=getch();
            if (c==75) x-=pas;
            if (c==77) x+=pas;
            if (c==72) y-=pas;
            if (c==80) y+=pas;
            cleardevice();
        }
    }
}
while (c!=27);
closegraph();
}

```

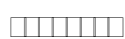
19.5. Instrucțiuni pentru desenare de figuri umplute

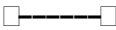
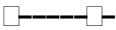
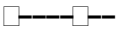
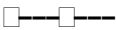
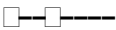
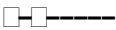

31. setfillstyle(model,c); setează modelul și culoarea de umplere ale figurilor pline. Valorile posibile pentru parametrul *model* sunt:

Denumire constantă	Valoare constantă	Descriere
EMPTY_FILL	0	Fără umplere
SOLID_FILL	1	Umplere uniformă
LINE_FILL	2	Hașură orizontală
LTSLASH_FILL	3	Hașură spre stânga cu linii subțiri
SLASH_FILL	4	Hașură spre stânga cu linii groase
BKSLASH_FILL	5	Hașură spre dreapta cu linii subțiri
LTBKSLASH_FILL	6	Hașură spre dreapta cu linii groase
HATCH_FILL	7	Hașură în cruce
XHATCH_FILL	8	Hașură în cruce obligă
INTERLEAVE_FILL	9	Hașură cu întretesare
WIDE_DOT_FILL	10	Umplere cu puncte rare
CLOSE_DOR_FILL	11	Umplere cu puncte dese
USER_FILL	12	Model definit de utilizator

32. setfillpattern(model,culoare); poate defini un nou model de umplere, unde *model* este un șir de 8 caractere, care definește o matrice 8 x 8 biți de valori 0 și 1.

Exemplu. Definirea următorului model de umplere.


 -> 11111111 (in baza 2) = FF (in baza 16) = 255 (in baza 10)

	-> 10000001 (in baza 2) = 81 (in baza 16) = 129 (in baza 10)
	-> 10000010 (in baza 2) = 82 (in baza 16) = 130 (in baza 10)
	-> 10000100 (in baza 2) = 84 (in baza 16) = 132 (in baza 10)
	-> 10001000 (in baza 2) = 88 (in baza 16) = 136 (in baza 10)
	-> 10010000 (in baza 2) = 90 (in baza 16) = 144 (in baza 10)
	-> 10100000 (in baza 2) = A0 (in baza 16) = 160 (in baza 10)
	-> 11000000 (in baza 2) = C0 (in baza 16) = 192 (in baza 10)

Observație: În exemplul de mai sus ‘-’ reprezintă spațiu, iar ‘□’ reprezintă punct.

```
int culoare=YELLOW;
char model[8]={255,129,130,132,136,144,160,192};
setfillpattern(model,culoare);
```

33. bar(x1,y1,x2,y2); se desenează un dreptunghi umplut cu culoarea și modelul curente, fără contur, având coordonate a două vârfuri diametral opuse ($x1,y1$) și ($x2,y2$).

34. bar3d(x1,y1,x2,y2,a,c); se desenează o bară tridimensională având coordonate a două vârfuri diametral opuse ($x1,y1$) și ($x2,y2$). Parametrul a reprezintă adâncimea barei (în pixeli), iar c specifică dacă bara are capac (când c este nenul) sau nu. Barele “fără capac” pregătesc supraetajare (aplicarea altor bare deasupra).

Exemplu. Sa se genereze 100 de cifre în baza 10. Să se reprezinte grafic numărul de apariții al fiecărei cifre folosind bare 3D.

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
# include <process.h>
# include <graphics.h>

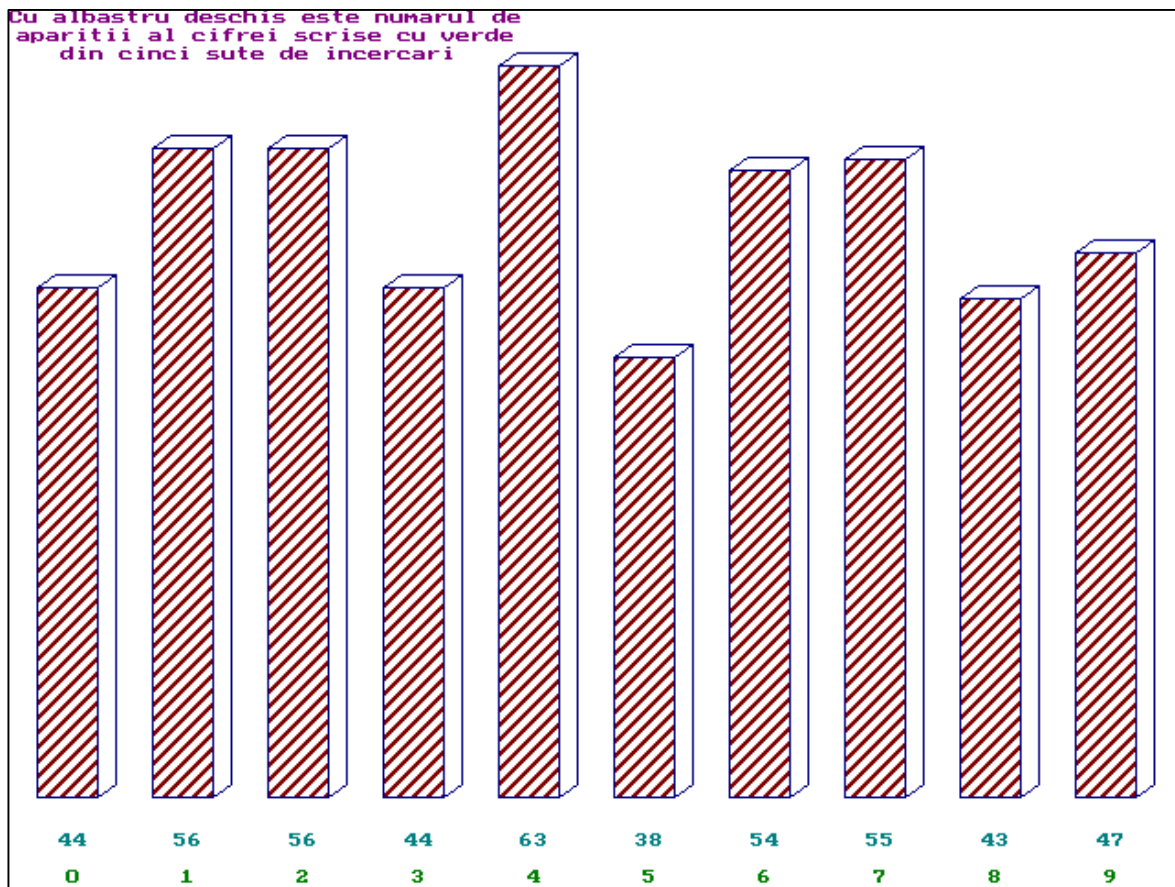
void main(void)
{
    int m=500,i,maxx,maxy,gd=DETECT,gm;
    char buff[10];
    long n[10],max;
    randomize();
    for (i=0;i<10;i++) n[i]=0;
    for (i=0;i<m;i++) n[random(10)]++; // numarul cifrelor alese aleator
    max=n[0];
    for (i=1;i<10;i++)
        if (n[i]>max) max=n[i];
    initgraph(&gd,&gm,"");
    setbkcolor(15);
    cleardevice();
    maxx=getmaxx()/10;
    maxy=getmaxy();
    setfillstyle(SLASH_FILL,RED);
    setcolor(MAGENTA);
    outtextxy(0,0,"Cu albastru deschis este numarul de");
    outtextxy(4,10,"aparitii al cifrei scrise cu verde");
    outtextxy(4,20,"din cinci sute de incercari");
    for (i=0;i<10;i++)
    {
```

```

setcolor (BLUE);
bar3d(i*maxx+maxx/4,maxy-50,(i+1)*maxx-maxx/4
      ,maxy-50-(long)(maxy-80)*n[i]/max,10,1);
setcolor (CYAN);
sprintf(buff,"%d",n[i]);
outtextxy(i*maxx+maxx/2-4,maxy-30,buff);
setcolor (GREEN);
sprintf(buff,"%d",i);
outtextxy(i*maxx+maxx/2,maxy-10,buff);
}
getch(); closegraph();
}

```

În urma execuției programului de mai sus, pe ecran se va afișa:



35. fillpoly(n,p); desenează un poligon umplut având și contur cu culorile și modelele curente, unde n este numărul de vârfuri al poligonului, iar în p se memorează coordonatele vârfurilor, la fel ca la **drawpoly**. Spre deosebire de **drawpoly**, funcția **fillpoly** închide poligonul.

36. fillellipse(x,y,rx,ry); desenează o elipsă umplută și cu contur, cu culorile și modelele curente având centrul în (x,y) și razele rx și ry .

37. pieslice(x,y,u1,u2,r); se desenează interiorul și conturul unui sector de cerc de rază r și de centru (x,y) , între unghiurile $u1$ și $u2$ (bucată dintr-o plăcintă), cu culorile și modelele curente.

Exemplu. Se citesc n valori întregi de la tastatura. Să se reprezinte grafic folosind sectoare de cerc.

```
#include<math.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<graphics.h>
#define R 230
#define PI 3.14159

int xc(float r,float u)
{
    int i=r*cos(u);
    return i;
}

int yc(float r,float u)
{
    int i=r*sin(u);
    return i;
}

void main(void)
{
    int i,n,gd=DETECT,gm,dx,dy,suma,a[100];
    float c1,c2,u,v;
    char s[5];
    clrscr();
    printf("Dati numarul de valori: ");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("Valoarea %2d: ",i+1);
        scanf("%d",&a[i]);
    }
    initgraph(&gd,&gm,"");
    dx=getmaxx()/2;
    dy=getmaxy()/2;
    suma=a[0];
    for (i=1;i<n;i++)
        suma=suma+a[i];
    setcolor(0);
    c2=0;
    u=360./suma;
    for (i=0;i<n-1;i++)
    {
        c1=c2;
        c2=c1+u*a[i];
        setfillstyle(1,15-i);
        pieslice(dx,dy,(int)c1,(int)c2,R);
        v=PI*(c1+c2)/360;
        sprintf(s,"%d",a[i]);
```

```

    outtextxy(dx+xc(2*R/3,v)-strlen(s)*4,getmaxy()-dy-yc(2*R/3,v),s);
}
c1=c2;
c2=c1+u*a[n-1];
setfillstyle(1,16-n);
pieslice(dx,dy,(int)c1,360,R);
v=PI*(c1+c2)/360;
sprintf(s,"%d",a[n-1]);
outtextxy(dx+xc(2*R/3,v)-strlen(s)*4,getmaxy()-dy-yc(2*R/3,v),s);
getch();
closegraph();
}

```

38. sector(x,y,u1,u2,rx,ry); desenează un sector de elipsă de raze rx și ry și de centru (x,y) , între unghiurile $u1$ și $u2$.

39. floodfill(x,y,c); umple folosind culoarea și modelul curente, interiorul unui contur având culoarea c , umplerea pornindu-se din punctul de coordonate (x,y) . Algoritmul de umplere cu *floodfill* este foarte lent.

Exemplu. Să se umple un contur aleator (aflat pe ecran) cu un model.

```

#include<math.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<graphics.h>
#define R 230
#define PI 3.14159

int xc(float r,float u)
{
    int i=r*cos(u); return i;
}

int yc(float r,float u)
{
    int i=r*sin(u); return i;
}

void main(void)
{
    int i,n,gd=DETECT,gm,dx,dy,suma,a[100];
    float c1,c2,u,v;
    char s[5];
    clrscr();
    printf("Dati numarul de valori: ");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("Valoarea %2d: ",i+1);
        scanf("%d",&a[i]);
    }
}

```

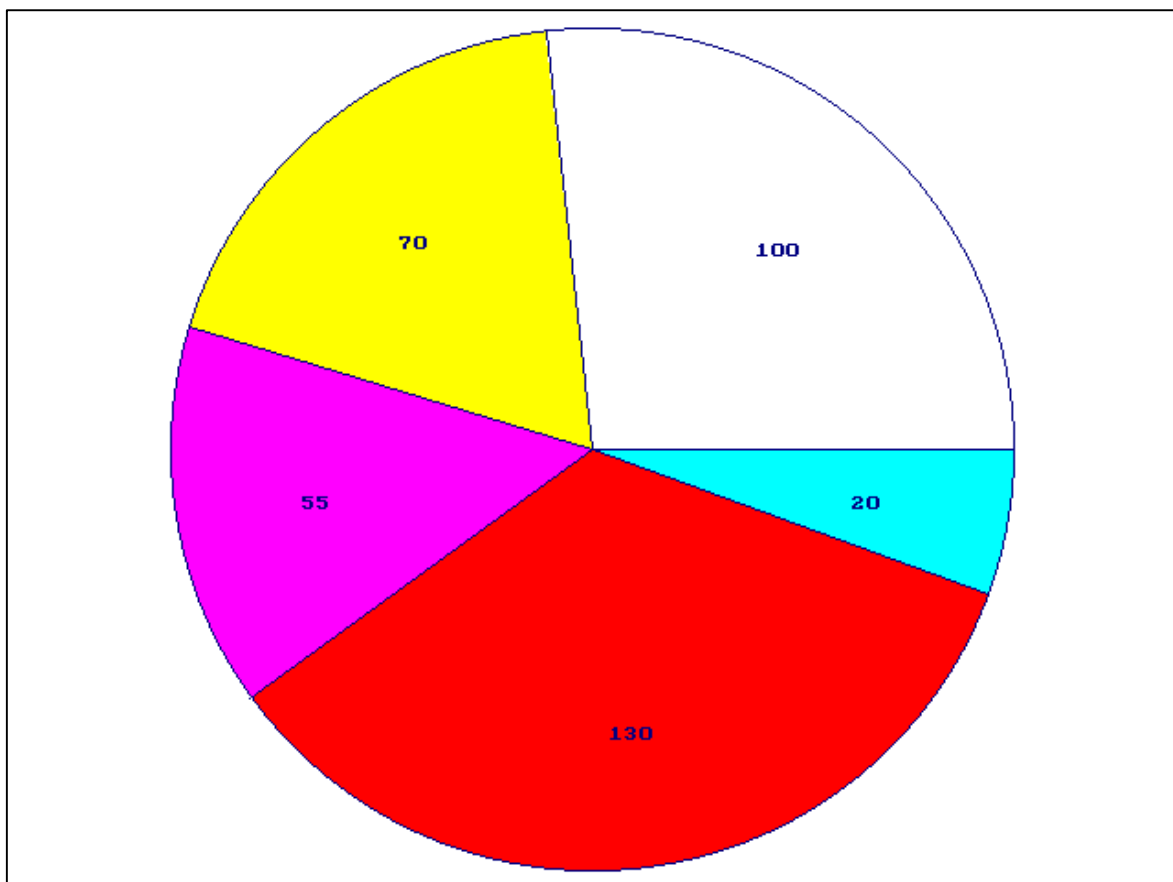


```

    }
    initgraph(&gd,&gm,"");
    cleardevice(); setcolor(BLUE);
    dx=getmaxx()/2; dy=getmaxy()/2;
    suma=a[0];
    for (i=1;i<n;i++) suma=suma+a[i];
    setbkcolor(15);
    c2=0; u=360./suma;
    for (i=0;i<n-1;i++)
    {
        c1=c2; c2=c1+u*a[i];
        setfillstyle(1,15-i);
        pieslice(dx,dy,(int)c1,(int)c2,R);
        v=PI*(c1+c2)/360;
        sprintf(s,"%d",a[i]);
        outtextxy(dx+xc(2*R/3,v)-strlen(s)*4,getmaxy()-dy-yc(2*R/3,v),s);
    }
    c1=c2; c2=c1+u*a[n-1];
    setfillstyle(1,16-n);
    pieslice(dx,dy,(int)c1,360,R);
    v=PI*(c1+c2)/360;
    sprintf(s,"%d",a[n-1]);
    outtextxy(dx+xc(2*R/3,v)-strlen(s)*4,getmaxy()-dy-yc(2*R/3,v),s);
    getch(); closegraph();
}

```

În urma execuției programului de mai sus, pe ecran se va afișa:



19.6. Instrucțiuni pentru paleta de culori

40. Citirea paletii de culori curente se face cu: **getpalette(&pal);**

Variabila *pal* este de tip structură de forma:

```
struct palettetype{
    unsigned char size;
    signed char colors[maxcolors+1];
};
```

41. Setarea paletii de culori se face cu funcția: **setallpalette(&pal);**

42. Modificarea unei singure culori din paleta se face cu: **setpalette(c1,c2);**. Culoarea *c1* este înlocuită cu culoarea *c2*.

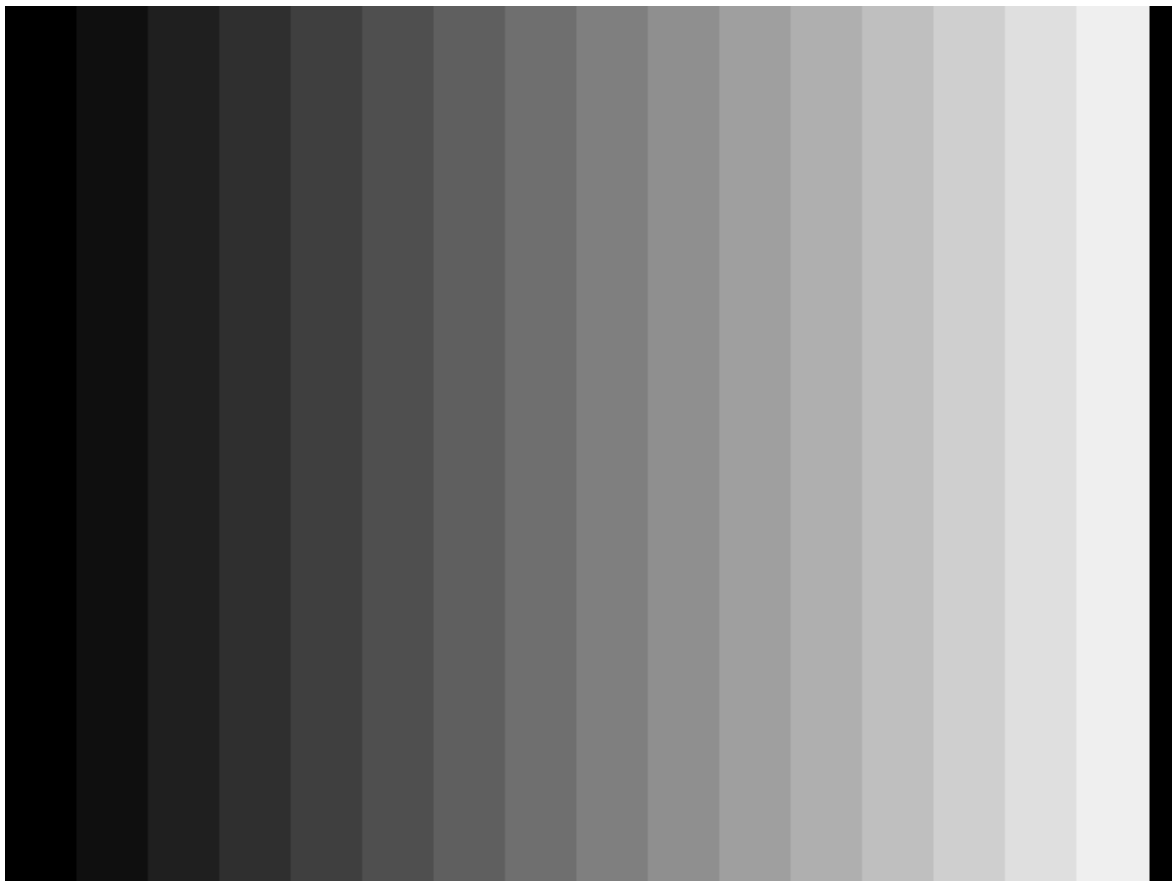
43. **setrgbpalette(c,R,V,B);** Culoarea din paleta cu codul *c* va avea nuanța obținută prin combinarea culorilor: roșu (de intensitate *R*) cu verde (de intensitate *G*) și cu albastru (de intensitate *B*). În C, intensitatea unei culori primare (roșu, verde sau albastru) poate lua valori între 0 și 63 (de la negru până la cea mai deschisă culoare).

Exemplu. Să se umple ecranul monitorului cu bare verticale în nuanțe degrade de gri (griul se obține combinând în aceeași măsură cele trei culori roșu, verde și albastru).

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

void main(void)
{
    int i,l,gd=DETECT,gm;
    struct palettetype pal;
    initgraph(&gd,&gm,"");
    getpalette(&pal); // citirea paletii
    for (i=0;i<pal.size;i++)
        setrgbpalette(pal.colors[i],i*4,i*4,i*4); // setarea culorii i
    l=getmaxx()/pal.size;
    for (i=0;i<pal.size;i++)
    {
        setfillstyle(SOLID_FILL,i);
        bar(i*l,0,(i+1)*l,getmaxy());
    }
    getch();
    closegraph();
}
```

În urma execuției programului de mai sus, pe ecran se va afișa:



19.7. Pagini grafice

Paginile grafice sunt zone distincte de memorie video. Dacă avem n pagini video, atunci acestea sunt numerotate $0, 1, \dots, n-1$.

Dacă modul grafic acceptă mai multe pagini grafice (cum ar fi modurile VGAMED sau VGALO), atunci pagina activă (cea pe care se desenează) poate fi separată de cea vizuală (cea care se vede). Este alocată cate o zonă distinctă de memorie video pentru fiecare pagină grafică. Oricare pagină poate fi setată ca fiind activă sau (și) vizuală, dar în orice moment există o singură pagină activă și o singură pagină vizuală (cele două pagini pot însă coincide). Printr-o succesiune de interschimbări de pagini grafice putem realiza o animație profesionistă eliminând fenomenul de clipire al ecranului monitorului la curățare și reafișare.

Dacă dispunem de cel puțin 2 pagini vizuale pentru animație procedăm astfel: desenăm pe una dintre pagini (cea activă), iar o alta pagină o lăsăm vizibilă pe ecran. Când am terminat de desenat pe pagina activă, o facem vizibilă, iar fosta pagină vizibilă devine activă, pe care urmează să desenăm următorul *frame*. Când aceasta este gata o facem vizibilă etc.

Un *frame* reprezintă o singură imagine din cele care compun animația.

Dacă imaginile ce compun animația nu sunt foarte complicate, putem obține un număr de până la 80 frame-uri pe secundă. O animație este considerată bună dacă avem minim 26 de frame-uri pe secundă (viteza obișnuită de redare a unui film).

Setarea unei pagini ca fiind activă sau vizibilă, se face cu ajutorul funcțiilor:

44. setactivepage(pag); setează pagina activă.

45. setvisualpage(pag); setează pagina vizuală.

Exemplu. Să se realizeze un joc în care în partea de jos a ecranului jucătorul să deplaseze stânga-dreapta o elipsă plină, iar sus să se miște aleator stânga-dreapta un dreptunghi umplut.

Jucătorul poate trage asupra inamicului (dreptunghiul de sus), care la rândul său lansează proiectile asupra elipsei. Se vor folosi pagini grafice.

```
# include <stdio.h>
# include <conio.h>
# include <graphics.h>
# include <process.h>
# include <stdlib.h>
# include <dos.h>
# define pas_inamic 5
# define pas_nava 10
# define pas_bomba1 10
# define pas_bomba2 5
# define l_nava 40
# define l_inamic 40

void ig()
{
    int err,gd=VGA,gm=VGAMED;
    initgraph(&gd,&gm,"");
    err = graphresult();
    if (err != grOk)
    {
        printf("Graphics error: %s\n", grapherrormsg(err));
        getch(); exit(1);
    }
}

int muta_inamic,x_inamic,y_inamic,x_bomba1,y_bomba1,x_bomba2,y_bomba2,
    x_nava,y_nava,maxx,maxy,pag=0,scor=0,timp=5000,vieti=5;

void inv_pag()
{
    setvisualpage(pag);
    if (pag) pag=0;
    else pag=1;
    setactivepage(pag);
}

void afis_nava()
{
    setfillstyle(SOLID_FILL,BLUE);
    setcolor(BLUE);
    fillellipse(x_nava+l_nava/2,y_nava,l_nava/2,10);
}

void afis_bomba1()
{
    if (y_bomba1<y_nava)
    {
        if (y_bomba1>pas_bomba1)
        {
            y_bomba1-=pas_bomba1;
        }
    }
}
```

```

        setcolor(CYAN);
        circle(x_bomba1,y_bomba1,3);
    }
else
    {
        if (x_bomba1>x_inamic && x_bomba1<x_inamic+l_inamic)
        {
            scor+=10;
            sound(800);
            delay(100);
            nosound();
        }
        y_bomba1=y_nava;
    }
}

void afis_bomba2()
{
    if (y_bomba2==y_inamic && random(10)==0 && x_inamic<x_nava+l_nava*2
        && x_inamic>x_nava-l_nava)
    {
        y_bomba2=y_inamic+pas_inamic;
        x_bomba2=x_inamic+l_inamic/2;
    }
    if (y_bomba2>y_inamic)
    {
        if (y_bomba2+pas_bomba2<y_nava)
        {
            y_bomba2+=pas_bomba2;
            setcolor(GREEN);
            circle(x_bomba2,y_bomba2,3);
        }
        else
        {
            if (x_bomba2>x_nava && x_bomba2<x_nava+l_nava)
            {
                vieti--;
                sound(1000);
                delay(200);
                nosound();
            }
            y_bomba2=y_inamic;
        }
    }
}

void afis_inamic()
{
    setfillstyle(SOLID_FILL,RED);
    bar(x_inamic,y_inamic,x_inamic+l_inamic,y_inamic+10);
}

```

```

void afis_scor()
{
    char s[10];
    gcvt(scor,5,s);
    setcolor(BLUE);
    outtextxy(0,0,"Scor:");
    outtextxy(50,0,s);
}

void afis_timp()
{
    char s[10];
    gcvt(timp,5,s);
    setcolor(MAGENTA);
    outtextxy(maxx-100,0,"Timp:");
    outtextxy(maxx-50,0,s);
}

void afis_vieti()
{
    char s[10];
    gcvt(vieti,5,s);
    setcolor(LIGHTBLUE);
    outtextxy(maxx-100,10,"Vieti:");
    outtextxy(maxx-40,10,s);
}

void main(void)
{
    char c='~';
    ig();randomize();
    maxx=getmaxx(); maxy=getmaxy();
    x_nava=maxx/2;
    y_nava=maxy-pas_bombal;
    y_bombal=y_nava;
    x_inamic=random(maxx-l_inamic);
    y_inamic=10;
    y_bomba2=y_inamic;
    setvisualpage(1); setactivepage(0);
    while (c!=27 && timp>0 && vieti>0)
    {
        delay(1); setbkcolor(15); cleardevice(); delay(1);
        afis_scor();
        afis_timp();
        afis_vieti();
        afis_nava();
        afis_inamic();
        afis_bombal();
        afis_bomba2();
        inv_pag();
        muta_inamic=(random(5)-2)*pas_inamic;
        if (x_inamic+muta_inamic<=maxx-l_inamic && x_inamic+muta_inamic>=0)
            x_inamic+=muta_inamic;
    }
}

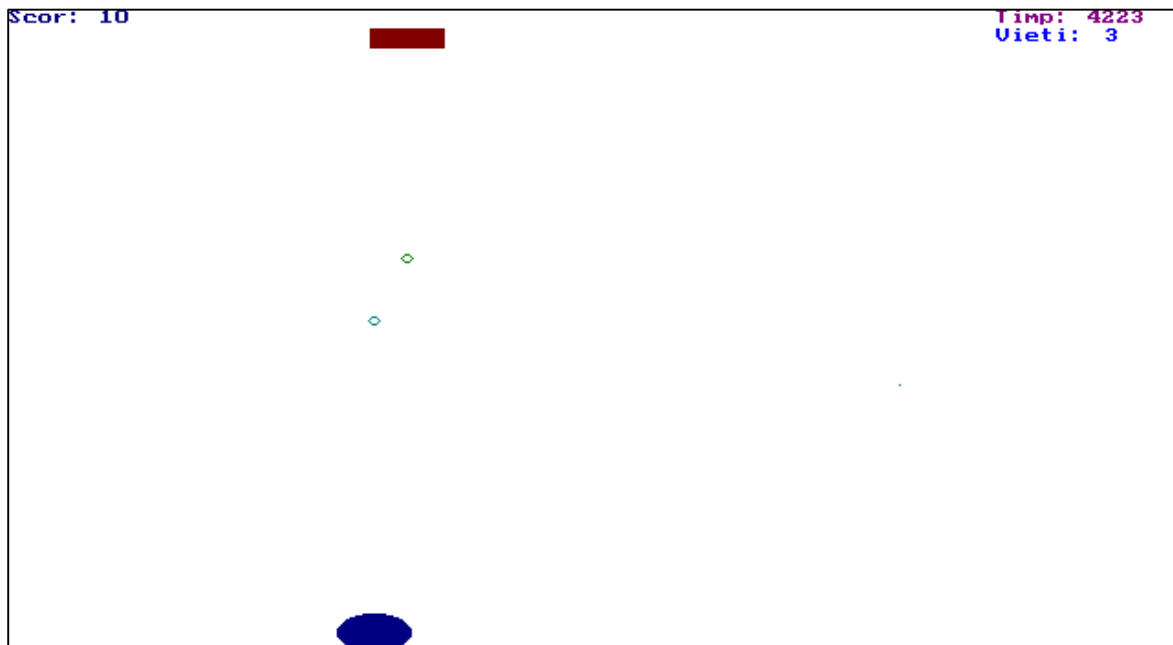
```

```

time--;
if (khit())
{
    c=getch();
    if (khit())
    {
        c=getch();
        if (c==77 && x_nava+pas_nava<=maxx-l_nava) x_nava+=pas_nava;
        if (c==75 && x_nava-pas_nava>=0) x_nava-=pas_nava;
    }
    if (c==13) getch();
    if (c==' ' && y_bombal==y_nava)
    {
        x_bombal=x_nava+l_nava/2;
        y_bombal=y_nava-pas_bombal;
    }
}
}
if (c==27) viet=0;
closegraph();
printf(" Scor:      %d\n",scor);
printf(" Bonus:      %d\n",viet*10);
printf("-----\n");
printf(" Scor final: %d\n",scor+viet*10);
getch();
}

```

Scena de desfășurare a jocului este:



19.8. Citire / scriere zonă de memorie video

46. `getimage(x1,y1,x2,y2,buf)`: Dreptunghiul cu vârful stânga-sus în (x1,y1) și vârful dreapta-jos în (x2,y2) este citit de pe pagina activă și depus în bufferul *buf*. Primii 4 octeți din *buf* sunt rezervați pentru dimensiunile imaginii citite.

47. putimage(x,y,buf,m); Imaginea conținută în bufferul *buf* este depusă pe pagina activă așa încât colțul stânga-sus să fie la coordonatele (x,y). Variabila *m* specifică modul în care se face afișarea. Modurile de copiere (pentru *m*) sunt:

Denumire constantă	Valoare constantă	Descriere
COPY_PUT	0	suprapune peste imaginea existentă
XOR_PUT	1	sau exclusiv între imagini
OR_PUT	2	sau între imagini
AND_PUT	3	și între imagini
NOT_PUT	4	copiază inversul imaginii

Exemplu. Să se simuleze mișcarea unei bile pe o masă de biliard care are aplicat un anumit model (de exemplu gen “tablă de șah”). Se vor folosi două pagini grafice, una pe care se află modelul mesei de biliard și cealaltă va fi vizuală. Mișcarea bilei se va realiza într-un ciclu care conține următorii pași: cu ajutorul funcțiilor *getimage* și *putimage* se citește modelul de pe pagina nevizuală și se afișează peste bilă, iar bila se va afișa la noile coordonate.

```
# include <stdio.h>
# include <conio.h>
# include <graphics.h>
# include <process.h>
# include <stdlib.h>
# include <dos.h>
# define r 5
# define r1 10
# define r2 8

void ig()
{
    int err, gd=VGA, gm=VGAMED;
    initgraph(&gd, &gm, "");
    err = graphresult();
    if (err != grOk)
    {
        printf("Graphics error: %s\n", grapherrormsg(err));
        getch();
        exit(1);
    }
}

void afisfundall(int cx1,int cy1,int cx2,int cy2)
{
    int i,j,x1,y1,x2,y2;
    x1=cx1/10;
    x2=cx2/10;
    y1=cy1/10;
    y2=cy2/10;
    for (i=x1;i<=x2;i++)
        for (j=y1;j<=y2;j++)
        {
            if ((i+j)%2) setfillstyle(1,14);
```



```

        else setfillstyle(1,LIGHTRED);
        if (i==x1 || i==x2 || j==y1 || j==y2) setfillstyle(1,LIGHTCYAN);
        bar(i*10,j*10,i*10+10,j*10+10);
    }
}

void afistaste()
{
    setcolor(BLUE);
    setfillstyle(1,DARKGRAY);
    bar(0,getmaxy()-45,145,getmaxy());
    rectangle(0,getmaxy()-45,145,getmaxy());
    setcolor(15);
    outtextxy(8,getmaxy()-40,"+/- Viteza");
    outtextxy(8,getmaxy()-30,"ENTER Pauza");
    outtextxy(8,getmaxy()-20,"SPATIU Repornire");
    outtextxy(8,getmaxy()-10,"ESC Exit");
}

void golbuff() { while (kbhit()) getch(); }

void main(void)
{
    char a[1000],b[1000],c;
    int i,dirx,diry,dx,dy,x,y,x1,y1,pauza=10,cx1,cx2,cy1,cy2,maxx,maxy;
    randomize();
    dirx=random(r*2+1)-r;
    diry=random(r*2+1)-r;
    if (dirx==0 && diry==0)
    {
        dirx=1;
        diry=-1;
    }
    dx=abs(dirx);
    dy=abs(diry);
    ig();
    setbkcolor(15);
    cleardevice();
    maxx=getmaxx()/2;
    maxy=getmaxy()/2;
    cx1=random(maxx-50);
    cx2=maxx+50+random(maxx-50);
    cy1=random(maxy-50);
    cy2=maxy+50+random(maxy-50);
    x=(cx1+cx2)/2;
    y=(cy1+cy2)/2;
    setactivepage(0);
    setvisualpage(1);
    afisfundall(cx1,cy1,cx2,cy2);
    setactivepage(1);
    afistaste();
    afisfundall(cx1,cy1,cx2,cy2);
    setcolor(1);

```

```

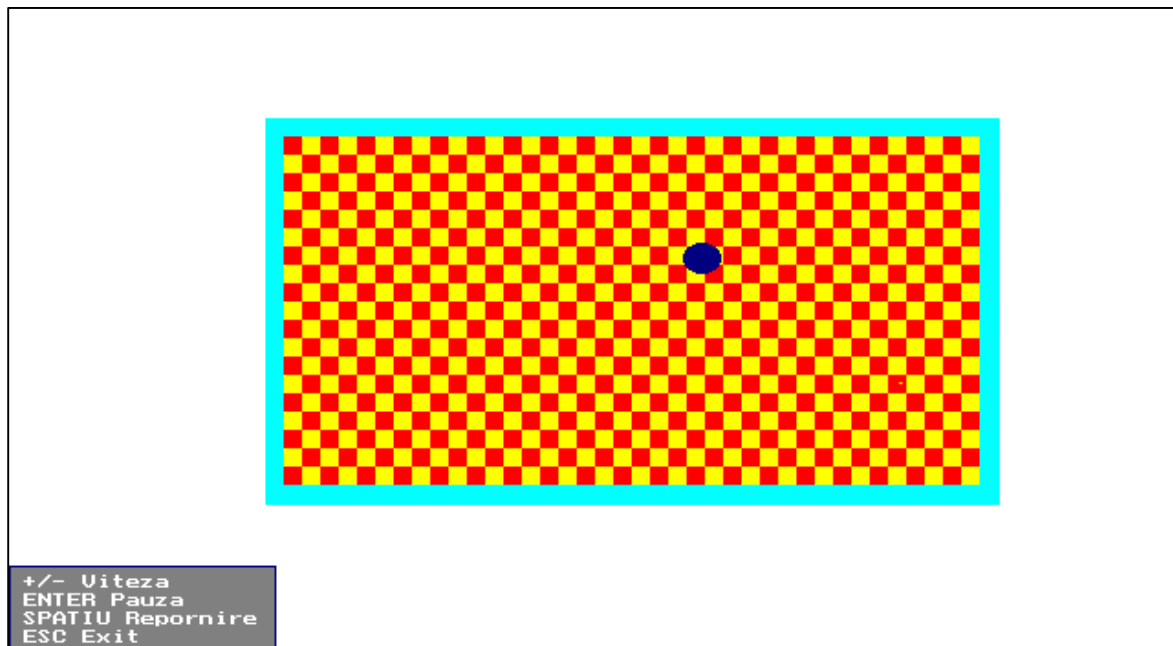
setfillstyle(1,1);
c='~';
while (c!=27)
{
    if (kbhit())
    {
        c=getch();
        if (c==13) getch();
        if (c=='+' && pauza>0) pauza--;
        if (c=='-' && pauza<50) pauza++;
        if (c==' ')
        {
            pauza=10;
            cx1=random(maxx-50);
            cx2=maxx+50+random(maxx-50);
            cy1=random(maxy-50);
            cy2=maxy+50+random(maxy-50);
            setactivepage(0);
            afisfundall(cx1,cy1,cx2,cy2);
            setactivepage(1);
            cleardevice();
            afistaste();
            afisfundall(cx1,cy1,cx2,cy2);
            setcolor(1);
            setfillstyle(1,1);
            x=(cx1+cx2)/2;
            y=(cy1+cy2)/2;
        }
        golbuff();
    }
    setactivepage(0);
    x1=x;
    y1=y;
    getimage(x-r1-dx,y-r2-dy,x+r1+dx,y+r2+dy,a);
    x+=dirx;
    y+=diry;
    if (x>cx2-r1-2*dx || x<cx1+r1+2*dx)
    {
        dirx=-dirx;
        sound(2000);
        delay(7);
        nosound();
    }
    if (y>cy2-r2-2*dy || y<cy1+r2+2*dy)
    {
        diry=-diry;
        sound(2000); delay(7); nosound();
    }
    fillellipse(x,y,r1,r2);
    getimage(x1-r1-dx,y1-r2-dy,x1+r1+dx,y1+r2+dy,b);
    putimage(x1-r1-dx,y1-r2-dy,a,0);
    setactivepage(1);
    putimage(x1-r1-dx,y1-r2-dy,b,0);
}

```

```

    delay(pauza);
}
closegraph();
}

```



19.9. Ferestre grafice

48. Definirea unei ferestre grafice se face cu **setviewport(x1,y1,x2,y2,m)**;

Efectul este asemănător cu instrucțiunea *window* din modul *text*. Afișarea se va face în interiorul ferestrei cu capătul stânga-sus în $(x1,y1)$ și capătul stânga-jos în $(x2,y2)$. Dacă m este diferit de 0, atunci desenele ce depășesc limitele ferestrei nu vor fi afișate. Dacă m este 0, atunci nu se va ține cont de fereastra definită.

49. Curățarea ferestrei grafice definite anterior cu *setviewport* se face cu **clearviewport()**;

Exemplu. Afișarea unui cerc care nu “încapă” într-un viewport.

```

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

int main(void)
{
    int gd=DETECT, gm, eroare;
    initgraph(&gd, &gm, "");
    setbkcolor(15);
    cleardevice();
    eroare=graphresult();
    if (eroare!=grOk)
    {
        printf("Graphics error: %s\n", grapherrormsg(eroare));
    }
}

```

```

    getch(); return 1;
}
setcolor(BLUE);
rectangle(100,50,500,400);
setcolor(RED);
setviewport(100,50,500,400,1);
circle(150,170,200);
getch(); closegraph(); return 0;
}

```

Rezumat

Pentru a desena sub MS-DOS în Borland C trebuie intrat în modul grafic folosind funcția **initgraph**. Pentru a se reveni în modul text se apelează funcția **closegraph()**. Antetele funcțiilor de lucru în modul grafic se găsesc în fișierul antet *graphics.h*.

Teme

1. Să se rescrie jocul prezentat într-unul dintre exemplele de mai sus așa încât acțiunea să se desfășoare în interiorul unui viewport.
2. Folosind anexa 1 a acestui curs să se deplaseze nava din jocul de mai sus folosind mouse-ul.

ANEXA 1 - Folosirea mouse-ului sub DOS

Mediul de programare Borland C/C++ pentru DOS nu oferă facilități directe de utilizare a mouse-ului. Propunem în continuare câteva funcții pentru utilizarea mouse-lui atât în modul text cât și în modul grafic.

Scrieți următorul cod într-un fișier cu numele *mouse.h*. Includeți acest fișier de fiecare dată când aveți nevoie să scrieți programe în care se folosește mouse-ul.

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

typedef struct graphtype
{
    char screenmask[16];
    char cursormask[16];
    int xactive,yactive;
}graphshapetype;

# define screenmask
{ \
    0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0x0001,\
    0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF,0xFEFF \
};

# define cursormask
{ \
    0x0100,0x0100,0x0100,0x0100,0x0100,0x0100,0x0100,0xFFFE,\
    0x0100,0x0100,0x0100,0x0100,0x0100,0x0100,0x0100,0x0000 \
};

# define waitdoubleclick 300;

typedef struct ppe
{
    char ofs,seg;
}ptrtype;

struct REGPACK r;

void clicknumber(int *buttons,int *clicks,int *x,int *y);
void defgraphlocator(graphshapetype shape);
void defsensitivity(int deltax,int deltay);
void deftextlocator(int styleflag,char scrmask,char cursmask);
void defxrange(int xmin,int xmax);
void defyrange(int ymin,int ymax);
void getbuttonpress(int *button,int *n,int *x,int *y);
void getbuttonrelease(int *button,int *n,int *x,int *y);
void getmotion(int *deltax,int *deltay);
void getmouse(int *button,int *x,int *y);
void hidelocator();
void resetmouse(int *foundmouse,int *buttons);
void setdoublespeed(int speed);
void setmouse(int x,int y);
```

```

void showlocator();
int foundmouse();

void resetmouse(int *foundmouse,int *buttons) // activeaza mouse-ul
{
    r.r_ax = 0;
    intr(0x33,&r);
    *buttons=r.r_bx;
    *foundmouse=(! r.r_ax==0);
}

void showlocator() // face sa apara cursorul mouse-lui
{
    r.r_ax=1;
    intr(0x33,&r);
}

void hidelocator() // ascunde cursorul mouse-lui
{
    r.r_ax=2;
    intr(0x33,&r);
}

void getmouse(int *button,int *x,int *y) // returneaza pozitia mouse-lui
{
    // si combinatia de butoane apasate
    r.r_ax=3;
    intr(0x33,&r);
    *button=r.r_bx;
    *x=r.r_cx;
    *y=r.r_dx;
}

void setmouse(int x,int y) // pozitioneaza mouse-ul pe ecran la coordonatele (x,y)
{
    r.r_ax=4;
    r.r_cx=x;
    r.r_dx=y;
    intr(0x33,&r);
}

void getbuttonpress(int *button,int *n,int *x,int *y)
{
    r.r_ax=5;
    r.r_bx=*button;
    intr(0x33,&r);
    *button=r.r_ax;
    *n=r.r_bx;
    *x=r.r_cx;
    *y=r.r_dx;
}

void getbuttonrelease(int *button,int *n,int *x,int *y) // returneaza butoanele apasate
{

```

```

    r.r_ax=6;
    r.r_bx=*button;
    intr(0x33,&r);
    *button=r.r_ax;
    *n=r.r_bx;
    *x=r.r_cx;
    *y=r.r_dx;
}

void clicknumber(int *buttons,int *clicks,int *x,int *y) // returneaza nr. de click-uri
{
    getmouse(buttons,x,y);
    if (*buttons==1)
    {
        delay(300);
        *buttons=0;
        getbuttonpress(buttons,clicks,x,y);
    }
    else *clicks=0;
}

void defxrange(int xmin,int xmax) // defineste limitele inferioare si
{                                     // superioare pe orizontala ecranului
    r.r_ax=7;
    r.r_cx=xmin;
    r.r_dx=xmax;
    intr(0x33,&r);
}

void defyrange(int ymin,int ymax) // defineste limitele inferioare si
{                                     // superioare pe verticala ecranului
    r.r_ax=8;
    r.r_cx=ymin;
    r.r_dx=ymax;
    intr(0x33,&r);
}

void defgraphlocator() // defineste cursorul în modul grafic
{
    r.r_ax=9;
    r.r_bx=1;//activ x
    r.r_cx=1;//activ y
    r.r_dx=0xfe;
    r.r_es=0x01;
    intr(0x33,&r);
}

void deftextlocator(int styleflag,char scmask,char cursmask) // defineste cursorul
{                                     // în modul text
    r.r_ax=10;
    if (styleflag) r.r_bx=0; else r.r_bx=1;
    r.r_cx=scmask;
    r.r_dx=cursmask;
}

```

```

    intr(0x33,&r);
}

void getmotion(int *deltax,int *deltay)    // returneaza pasul de miscare
{
    // pe orizontala si pe verticala
    r.r_ax=11;
    intr(0x33,&r);
    *deltax=r.r_cx;
    *deltay=r.r_dx;
}

void defsensitivity(int deltax,int deltay) // defineste sensibilitatea la miscare
{
    // pe orizontala si pe verticala
    r.r_ax=15;
    r.r_cx=deltax;
    r.r_dx=deltay;
    intr(0x33,&r);
}

void setdoublespeed(int speed)
{
    r.r_ax=19;
    r.r_dx=speed;
    intr(0x33,&r);
}

```

Ca aplicație la utilizarea mouse-ului în modul grafic propunem desenarea de cercuri, pătrate și elipse la apăsarea butoanelor stânga, dreapta, respectiv stânga împreună cu dreapta (simultan):

```

#include <conio.h>
#include <string.h>
#include <graphics.h>
#include "mouse.h" // includere fisier cu functiile pentru mouse de mai sus

#define r 40 // dimensiune figuri (cercuri si patrate)

void main(void)
{
    char s[10];
    int gd=DETECT,gm,buton,x,y,dx,dy;
    initgraph(&gd,&gm,"");
    showlocator(); // face sa apara cursorul mouse-lui
    do
    {
        getmouse(&buton,&x,&y);    // returnare buton si pozitie mouse
        getmotion(&dx,&dy);        // returnare valori deplasare mouse
        if (dx || dy)              // verificare daca s-a miscat mouse-ul
        {
            setfillstyle(SOLID_FILL,RED);
            setcolor(WHITE);
            bar(0,0,56,10);
            sprintf(s,"%3d/%3d",x,y);

```



```

        outtextxy(1,2,s); // afisare pozitie cursor mouse
    }
    switch (buton)
    {
        case 1: // click buton stanga mouse
            setcolor(YELLOW);
            circle(x,y,r);
            break;
        case 2: // click buton dreapta mouse
            setcolor(LIGHTCYAN);
            rectangle(x-r,y-r,x+r,y+r);
            break;
        case 2: // click butoane stanga+dreapta mouse
            setcolor(LIGHTGREEN);
            ellipse(x,y,0,360,r,2*r);
            break;
    }
}
while (!kbhit());
getch(); closegraph();
}

```

Aplicația pe care o propunem pentru utilizarea mouse-ului în modul text este afișarea caracterelor x și o la apăsarea butoanelor stânga, respectiv dreapta:

```

#include <conio.h>
#include <stdio.h>
#include "mouse.h" // includere fisier cu functiile pentru mouse de mai sus

void main(void)
{
    int buton,x,y,dx,dy;
    textbackground(0); clrscr();
    showlocator(); // face sa apara cursorul mouse-lui
    do
    {
        getmouse(&buton,&x,&y); // returnare buton si pozitie mouse
        getmotion(&dx,&dy); // returnare valori deplasare mouse
        if (dx || dy) // verificare daca s-a miscat mouse-ul
        {
            textcolor(WHITE);
            textbackground(RED);
            gotoxy(1,1);
            cprintf("%3d/%3d",x,y); // afisare pozitie cursor mouse
        }
        switch (buton)
        {
            case 1: // click buton stanga mouse
                textbackground(0);
                textcolor(YELLOW);
                gotoxy(x/8+1,y/8+1);
                cprintf("o");
                break;
            case 2: // click buton dreapta mouse

```

```

        textbackground(0);
        textcolor(LIGHTCYAN);
        gotoxy(x/8+1,y/8+1);
        cprintf("x");
        break;
    }
}
while (!kbhit()); // cand se apasa buton de la tastatura se paraseste programul
getch();
}

```

Observație: Poziția mouse-ului în modul text este dată de către funcția *getmouse* tot în puncte (ca și în modul grafic). Rezoluția ecranului în modul text obișnuit *co80* este 640x200. De aceea, pentru a afla poziția mouse-ului în coordonate text, trebuie ca la poziția în puncte împărțită la 8 să se adauge 1. Astfel, obținem coordonatele text ale cursorului mouse-ului $(X,Y) = (x/8+1,y/8+1)$. Evident obținem că $X \in \{1, 2, \dots, 80\}$ și $Y \in \{1, 2, \dots, 25\}$, pornind de la coordonatele în puncte (x,y) returnate de funcția *getmouse*, unde $x \in \{0, 8, 16, \dots, 632\}$ și $y \in \{0, 8, 16, \dots, 192\}$.

Scrieți un program C în care se citesc coordonatele vârfurilor unui poligon. Translați și rotiți poligonul pe ecran cu ajutorul mouse-ului.

ANEXA 2 - Urmărirea execuției unui program. Rulare pas cu pas.

Pentru a vedea efectiv traseul de execuție și modul în care se își modifică variabilele valorile într-un program, putem rula pas cu pas. Acest lucru se face în Borland C/C++ cu ajutorul butoanelor **F7** sau **F8**, iar în Visual C++ cu **F10** sau **F11**, combinații de taste care au ca efect rularea liniei curente și trecerea la linia următoare de execuție.

Execuția programului până se ajunge la o anumită linie se face apăsând pe linia respectivă butonul **F4** în Borland C/C++ și respectiv **Ctrl+F10** sau **Ctrl+F11** în Visual C++.

În Borland C/C++ pentru a fi posibilă urmărirea execuția unui program, în meniul Options, la Debugger, trebuie selectat On în Source Debugging ! În lipsa acestei setări, dacă se încearcă execuția pas cu pas, se afișează mesajul de atenționare **WARNING: No debug info. Run anyway?**. Este bine ca la *Display Swapping* (în fereastra *Source Debugging*) să se selecteze opțiunea *Always*, altfel fiind posibilă alterarea afișării mediului de programare. Reafișarea mediului de programare se poate face cu *Repaint desktop* din meniul \equiv . Este bine de știut că informațiile legate de urmărirea execuției sunt scrise în codul executabil al programului ceea ce duce la o încărcare inutilă a memoriei când se lansează în execuție aplicația. Așa că programul, după ce a fost depanat și este terminat, este indicat să fie compilat și link-editat cu debugger-ul dezactivat.

Pentru ca execuția programului să se întrerupă când se ajunge pe o anumită linie (*break*), se apasă pe linia respectivă **Ctrl+F8** în Borland C/C++ și **F9** în Visual C++. Linia va fi marcată (de obicei cu roșu). Pentru a anula un *break* se apasă tot **Ctrl+F8**, respectiv **F9** pe linia respectivă.

Execuția pas cu pas a unui program poate fi oprită apăsând **Ctrl+F2** în Borland C/C++ și **Shift+F5** în Visual C++.

Dacă se dorește continuarea execuției programului fără *Debugger*, se poate apăsa **Ctrl+F9** în Borland C/C++ și **F5** în Visual C++.

În orice moment, în Borland C/C++ de sub DOS rezultatele afișate pe ecran pot fi vizualizate cu ajutorul combinației de taste **Alt+F5**.

În Borland C/C++ valorile pe care le iau anumite variabile sau expresii pe parcursul execuției programului pot fi urmărite în fereastra **Watch**, pe care o putem deschide din meniul *Window*. Adăugarea unei variabile sau a unei expresii în **Watch** se face apăsând **Ctrl+F7**, sau **Insert** în fereastra **Watch**. Dacă se apasă **Enter** pe o expresie sau variabilă din fereastra **Watch**, aceasta poate fi modificată.

În Visual C++ valoarea pe care o are o variabilă pe parcursul urmăririi execuției unui program poate fi aflată mutând cursorul pe acea variabilă. Urmărirea valorii unei expresii pe parcursul rulării programului pas cu pas se poate face din meniul **Debug** folosind comanda **QuickWatch**. Adăugarea unei expresii într-o listă de urmărire se face apăsând butonul **AddWatch**.

BIBLIOGRAFIE

1. A. Deaconu, *Programare avansată în C și C++*, Editura Univ. "Transilvania", Brașov, 2003.
2. J. Bates, T. Tompkins, *Utilizare Visual C++ 6*, Editura Teora, 2000.
3. Nabajyoti Barkakati, *Borland C++ 4. Ghidul programatorului*, Editura Teora, 1997.
4. B. Stroustrup, *The C++ Programming Language*, a doua ediție, Addison-Wesley Publishing Company, Reading, MA, 1991..
5. T. Faison, *Borland C++ 3.1 Object-Oriented Programming*, ediția a doua, Sams Publishing, Carmel, IN, 1992.
6. R. Lafore, *Turbo C++ - Getting Started*, Borland International Inc., 1990.
7. R. Lafore, *Turbo C++ - User Guide*, Borland International Inc., 1990.
8. R. Lafore, *Turbo C++ - Programmer's Guide*, Borland International Inc., 1990.
9. O. Catrina, I. Cojocaru, *Turbo C++*, ed. Teora, 1993.
10. M. A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, MA, 1990.
11. S. C. Dewhurst, K. T. Stark, *Programming in C++*, Prentice Hall, Englewood Cliffs, NJ, 1989.
12. E. Keith Gorlen, M. Sanford, P. S. Plexico, *Data Abstraction and Object-Oriented-Programming in C++*, J. Wiley & Sons, Chichester, West Sussex, Anglia, 1990.
13. B. S. Lippman, *C++ Primer*, ediția a doua, Addison-Wesley, Reading, MA, 1991.
14. C. Spircu, I. Lopatan, *Programarea Orientată spre Obiecte*, ed. Teora.
15. M. Mullin, *Object-Oriented Program Design with Examples în C++*, Addison-Wesley, Reading, MA, 1991.
16. I. Pohl, *C++ for C Programmers*, The Benjamin/Cummings Publishing Company. Redwood City, CA, 1989.
17. T. Swan, *Learning C++*, Sams Publishing, Carmel, IN, 1992.
18. K. Weiskamp, B. Flaming, *The Complete C++ Primer*, Academic Press, Inc., San Diego, CA, 1990.
19. J. D. Smith, *Reusability & Software Construction: C & C++*, John Wiley & Sons, Inc., New York, 1990.
20. G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company. Redwood City, CA, 1991.

21. B. Meyer, *Object-Oriented Software Construction*, Pretice Hall International (U.K.) Ltd. Hertfordshire, Marea Britanie, 1988.
22. L. Pinson, R. S. Wiener, *Applications of Object-Oriented Programming*, Addison-Wesley Publishing Company, Reading, MA, 1990.
23. J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, Englewood-Cliffs, NJ, 1991.
24. L. A. Winblad, S. D. Edwards, D. R. King, *Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA, 1990.
25. R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.
26. D. Claude, *Programmer en Turbo C++*, Eyrolles, Paris, 1991.