Adrian DEACONU

PROGRAMARE PROCEDURALĂ

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 şi următoare.

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

Prezentăm în continuare o posibilă împărţire a lucrării în 14 unităţi de curs:

Cursul 1. Structura unui program C (capitolul 1)

Cursul 2. Tipuri numerice de date (capitolul 2)

Cursul 3. Funcţii de scriere şi citire în C (capitolul 3)

Cursul 4. Instrucţiuni de decizie (capitolul 4)

Cursul 5. Instrucţiuni repetitive(capitolul 5)

Cursul 6. Tipul char (capitolul 6)

Cursul 7. Pointeri, tablouri de elemente (capitolul 7)

Cursul 8. Funcţii în C (capitolul 8)

Cursul 9. String-uri (capitolul 9)

Cursul 10. Structuri (capitolul 10)

Cursul 11. Pointeri către structuri (capitolul 11)

Cursul 12. Fişiere în C (capitolul 12)

Cursul 13. Variabile statice (capitolul 13)

Cursul 14. Funcţii cu listă variabilă de argumente (capitolul 14)

CUPRINS

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 şi s-a bucurat încă de la început de un real succes datorită uşurinţei cu care se puteau scrie programe.

La sfârşitul anilor 1980 a apărut limbajul C++ ca o extensie a limbajului C. C++ preia facilităţile oferite de limbajul C şi aduce elemente noi, dintre care cel mai important este noţiunea 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 şi 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 şi 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 şi mediile Turbo C++ Professional şi 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 şi C++ dintr-o perspectivă atât Borland, cât şi Microsoft. Pe alocuri vor fi punctate micile diferenţe dintre cele două compilatoare.

1. Structura unui program C

Obiective

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

Un program C se salvează de obicei cu extensia *C* şi, 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 obişnuit 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 şi a constantelor globale este opțională. Mai mult, pot apărea mai multe grupuri de includeri, macrocomenzi, definiții de tipuri de date, variabile şi 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 `/*` şi `*/` în C se pun comentariile (aşa 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 afişează un mesaj de salut:

```
# include <stdio.h> /* includerea fişierului 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 şi 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 şi obțineam un mesaj de eroare.

1.1. Includeri

La începutul unui program C sau C++ se includ de obicei fişiere antet. Un fişier antet se recunoaşte uşor prin faptul că are de obicei extensia *h*. Se pot include şi fişiere cu extensia *C* sau *CPP*, care conțin cod C, respectiv cod C++.

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

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

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

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

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

Pentru limbajul C o parte dintre funcții sunt incluse implicit (compilatorul C le cunoaşte fără a fi nevoie includerea vreunui fişier). Totuşi, 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* (şi implicit se foloseşte compilatorul C), atunci includerea fişierului antet *stdio.h* nu este neapărat necesară. Dacă salvăm însă fişierul cu extensia *CPP*, atunci includerea lui *stdio.h* este obligatorie.

Iată în final şi câteva exemple de includeri:

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

1.2. Constante. Macrocomenzi.

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

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

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

Iată şi câteva exemple:

```
const int x=1+4;
const a=1,b=2*2; /* tot constante intregi ! */
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 obişnuite, *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 cunoaşte 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, deşi 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 şi 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 şi nu la expresia *(1+2)*5* cu valoarea 15.

În exemplul de mai sus la ultima macrocomandă s-a folosit operatorul ## care alipeşte (concatenează) doi tokeni. Astfel, un apel de forma *alipire(x,yz)* se înlocuieşte 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 înlocuieşte 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 creşte.

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* si *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 şi clase şablon pentru care unul sau mai multe tipuri de date sunt nespecificate, identificarea acestor tipuri făcându-se în momentul compilării programului. Şabloanele reprezintă un suport pentru programare generică în adevăratul sens al cuvântului. Asupra şabloanelor 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 şi fără cuvântul *struct*, fără o definire prealabilă cu *typedef*.

Asocierile de nume unor tipuri de date se pot face atât în exteriorul, cât şi 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 şi 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 greşit şi 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 şi 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 fişier etc.), programul se părăseşte 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* şi reține numărul de parametri în linie de comandă cu care s-a executat programul, iar al doilea este un şir 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 narg,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:

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

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

Rezumat

Înaintea funcției *main* pot apărea niciuna, una sau mai multe secțiuni de tipul:

- **includeri.** Pot fi incluse în general fişiere antet (cu extensia **.h**), dar pot apărea şi fişiere cu extensia **.c** sau **.cpp**. Aceste fişiere conțin în general antete (definiții) de funcții, definiții de tipuri de date şi constante, macrocomenzi, dar pot apărea şi 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 incepe cu cuvântul rezervat **typedef**.
- **declarații de variabile.**
- **definiții de constante globale.** Constantele în C se definesc după cuvântul rezervat **const**. O constantă obişnuită poate fi definită şi ca o macrocomandă.
- **definiții sau/şi descrieri de funcții.** Funcțiile programului C se declară deasupra funcției main şi 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 maximul dintre două numere.
2. Scrieți o macromandă pentru maximul dintre trei valori.
3. Folosind prima macrocomandă scrieți o macromandă pentru maximul dintre patru valori.
4. Ce trebuie să conțină parametrii **narg** şi **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 cunoştiință cu tipurile numerice de date ale limbajului C, modul lor de reprezentare în memorie, domeniul de valori, operatorii specifici acestor tipuri de date precum şi cu alți operatori pe care îi întâlnim în C.

În C şi C++ avem două categorii de tipuri numerice de date: tipuri întregi şi 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) şi fără semn (unsigned) din C. Facem observația că numărul de octeți pe care se reprezintă în memorie valorile întregi şi 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:

i. În Visual C tipul *enum* coincide cu tipul *short int*, iar tipul *int* coincide cu *long*.

ii. Tot ceea ce apare între paranteze rotunde în tabelul de mai sus este opțional.
iii. Tipurile numerice de date au o dublă utilizare în C şi C++: pentru valori numerice (numere) şi 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*)
iv. Tipurile *char* şi *unsigned char* în C şi C++ au o triplă întrebuințare: pentru valori întregi pe un octet, pentru valori booleene şi 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 acelaşi lucru cu 97 !
v. 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 şi 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**:** ++ şi --. 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 şi apoi noua valoare a lui y se atribuie lui x şi, în consecință, x devine 2. Dacă operatorul ++ se foloseşte în forma postincrementată, adică x = y++, atunci lui x întâi i se atribuie valoarea variabilei y şi 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 şi ++, tot sub două forme, predecrementare şi 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 şi numai dacă *x* este nenul.

Operatori la nivel de bit aplicabili numai pe valori întregi sunt: & (şi pe biți), | (sau pe biți), ^ (sau exclusiv), >> (*shift*-are biți la dreapta), << (*shift*-are biți la stânga), ~ (not pe biți). Pentru a înțelege cum funcționează aceşti operatori dăm următorul exemplu:

Fie *a* şi *b* două variabile întregi pe un octet, *a* reprezentat pe biți este (1,0,1,1,0,1,1,1) şi 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 ^ 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), ~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 şi 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 şi variantele combinate cu atribuire: &=, |=, ^=, <<=, >>=, 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 şi 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 şi 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* şi 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** şi de **incrementare**, respectiv **decrementare** funcționează şi pentru valori reale.

2.5. Alți operatori în C

Operatorii logici: && (şi), || (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 şi se foloseşte 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* primeşte valoarea *val2*.

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

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

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 şi 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.

Operatorul , (virgulă) se aplică între două expresii care se evaluează după regula: se evaluează întâi prima expresie şi 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: $expr_1, expr_2, \ldots , expr_n$, 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 şi unsigned long**. De departe cel mai cunoscut şi utilizat tip de date din C este *int*, care se reprezintă pe doi sau patru octeți, în funcție de compilator şi de sistemul de operare. Operatorii aritmetici binari pentru valori întregi sunt: + (adunare), - (scădere), * (înmulțire), / (câtul împărțirii) şi % (restul împărțirii).

Tipurile întregi de date pot fi folosite şi 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** şi **long double**. Operatorii aritmetici binari pentru valori reale sunt: + (adunare), - (scădere), * (înmulțire) şi / (împărțire).

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

Temă

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

3. Funcții de scriere şi citire în C

Obiective

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

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

3.1. Funcția printf

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

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

Funcția *printf* afişează pe ecranul monitorului *constanta_sir_de_caractere* în care toți descriptorii de format se în locuiesc 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 şi inițializate (aşa cum am procedat în exemplul de mai sus pentru *n* şi *x*).

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

Înlocuirea descriptorilor în constanta de tip şir 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 înlocuieşte 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 afişarea va fi eronată!

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

%c – caracter (*char*)
%s – şir de caractere încheiat cu caracterul '\0' (string)
%d – întreg cu semn (*int*)
%u – întreg fără semn (*unsigned int*)
%ld – întreg lung (pe 4 octeți) cu semn (*long*)
%lu – întreg lung (pe 4 octeți) fără semn (*unsigned long*)
%x – întreg în baza 16 fără semn (*int*) (cifrele în hexazecimal pentru *10*, *11*, *12*, *13*, *14*, *15* sunt litere mici, adică *a*, *b*, *c*, *d*, *e* şi *f*)
%X – la fel ca %x, numai ca cifrele în hexazecimal pentru sunt litere mari
%o – întreg în baza 8 fără semn (*int*)
%f – real pe 6 octeți (*float*), notație zecimală (fără exponent)
%e – real pe 6 octeți (*float*), notație exponențială, ştiințifică (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 (*float*), notație zecimală sau exponențială, care este mai scurtă, iar dacă se afişează exponențial, atunci litera de la exponent este *e*

%G – real pe 6 octeți (*float*), 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 (*double*), notație zecimală
%le – real pe 8 octeți (*double*), notație exponențială (litera exponent *e* este mică)
%lE – la fel ca la %le, numai litera exponent *E* este mare
%lg – real pe 8 octeți (*double*), notație zecimală sau exponențială, care e mai scurtă, dacă e cazul se foloseşte literă mică pentru exponent
%lG – real pe 8 octeți (*double*), notație zecimală sau exponențială, care e mai scurtă, dacă e cazul se foloseşte literă mare pentru exponent
%Lf – real pe 10 octeți (*long double*), notație zecimală
%Le – real pe 10 octeți (*long double*), notație exponențială (litera de la exponent este mică, adică *e*)
%LE – real pe 10 octeți (*long double*), 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 afişarea 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 afişarea unei valori, tipul alinierii, caracterele ce se completează în locurile libere (spații, zerouri) etc. Dăm în acest sens câteva exemple:

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

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

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

%010.2f realizează afişarea 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 afişa 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 (afişarea unui întreg şi a unui număr real), după afişarea pe ecran a mesajului, se sare la începutul liniei următoare, deoarece şirul s-a încheiat cu *\n*. Combinația de caractere *\n* este una dintre aşa 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 – afişarea 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 şir constant de caractere, semnul \ (*backslash*) trebuie dublat. Asta se întâmplă mai ales când este vorba de căi de fişiere. De asemenea, semnele *“* (*ghilimele*) şi *‘* (*apostrof*) trebuiesc precedate de semnul \ (*backslash*).

În C pentru a trimite un mesaj în fluxul standard de erori *stderror* 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 aşa cum face *printf*, în exemplele pe care o să le dăm în această carte vom folosi funcția *printf* pentru afişarea 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 şir de caractere, aşadar nu este necesar să punem semnul *&* (şi) în fața lui *s* la citire. Dacă se pune totuşi semnul *&* în fața variabilei *s* la citire, nu se semnalează eroare nici măcar atenționare, “pleonasmele” în C în general sunt ignorate de compilator. Vom reveni cu mai multe explicații când vom vorbi despre pointeri şi tablouri (şiruri).

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

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

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

Asupra acestor funcţii o să revenim atunci când o să discutăm despre string-uri, respectiv fişiere.

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

Trebuie precizat faptul că funcţia *printf* nu ţine cont de marginile ferestrei text definite cu *window*. Ca parametri, funcţia *window* primeşte cele 4 coordonate ecran text ale colţurilor stânga-sus şi respectiv dreapta-jos ale ferestrei.

Rezumat

Afişarea formatată a mesajelor pe ecran se face cu ajutorul funcţiei **printf**, iar preluarea formatată a datelor de la tastatură se face folosind funcţia **scanf**. Pentru a utiliza aceste funcţii trebuie să includem fişierul antet **stdio.h**.

Temă

Afişaţi pe ecranul monitorului cele două tabele cu tipurile numerice de date de la capitolul anterior.

4. Instrucţiuni de decizie

Obiective

Ne propunem să vedem cum se redacteză o instrucţiune de decizie în C.

În C există două instrucţiuni de decizie: *if ... else* şi instrucţiunea de decizie multiplă (cu oricâte ramuri) *switch ... case*.

4.1. Instrucţiunea if

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

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

Semnificaţia instrucţiunii este: dacă este adevărată condiţia, atunci se execută *grupul de instrucţiuni 1*, altfel se execută *grupul de instrucţiuni 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 sa 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 trebuiesc delimitate cu acolade. Nu este greşit însă dacă le delimităm totuşi cu acolade.

2) După fiecare grup de instrucțiuni punem în general instrucțiunea *break*. În lipsa instrucțiunii *break* se execută şi instrucțiunile de pe ramurile de mai jos până la sfârşitul instrucțiunii *switch* (inclusiv cele de pe ramura *default*) sau până se întâlneşte primul *break*. Instrucțiunea *break* întrerupe execuția instrucțiunii *switch* şi a celor repetitive (*for*, *while* şi *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 şi distincte două câte două.

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

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

```
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) şi instrucțiunea **switch** (cu oricâte ramuri). Expresia după care se face selecția ramurii **case** trebuie să aibă o valoare întreagă. De asemenea, de reținut este şi faptul că în principiu, fiecare ramură a instrucțiunii *switch* se termină cu un apel **break**.

5. Instrucțiuni repetitive

Obiective

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

În C există trei instrucțiuni repetitive: *for*, *while* şi *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) şi 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("Cmmmc(%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 opreşte 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 **s**ortarea crescătoare unui şir 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 intrucțiuni repetitive: **for, while** şi **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 şi 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 citeşte de la tastatură un număr întreg. Să se verifice dacă este prim.
2. Afişați primele *n* numere naturale prime, unde *n* este citit de la tastatură.
3. Se citeşte 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 aceleaşi cu situația în care le citim de la dreapta spre stânga.

6. Tipul char

Obiective

Ne propunem să vedem unde găsim şi 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 tabelei ASCII. De acest lucru o să ne ocupăm în acest capitol.

Definițiile principalelor funcții de lucru cu caractere în C le găsim în fişierul antet *ctype.h*. Dintre aceste funcții o să le prezentăm numai pe cele mai importante şi mai utile. Facem observația că toate funcțiile de mai jos returnează o valoare de tip *int* şi 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.

Şi în fişierul antet *conio.h* întâlnim câteva funcții legate de tipul *char*:

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

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

3) *putch(c)* afişează 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 apăsat un buton de la tastatură. Dacă buffer-ul este nevid, atunci se returnează o valoare nenulă.

Programul următor afişează 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 fişierul **ctype.h**.

Temă

Să se afişeze pe ecran cele 256 caractere ale tabelei 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 şir 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** şi memorează o adresă de memorie, la care este reținută o informație de tipul *tip*. O adresă se memorează pe 4 octeți şi 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 & şi & anulează pe *, adică **&var* este totuna cu *var*, iar *&*numepointer* este acelaşi lucru cu *numepointer*.

Pentru valori de tip pointer funcționează operatorii: =, +, -, +=, -=, ++, -- şi !. 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 şi 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 şi 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 acelaşi număr de octeți.

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

Pentru un vector putem aloca 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* şi *lungimev-1*. De exemplu, vectorul *a* definit mai sus are elementele: *a[0], a[1], ... , a[9]*.

Dacă se trece de capetele *0* şi *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 greşit!

În declarația *numevector[lungimev]*, *numevector* este un pointer care reține adresa de început a zonei de memorie alocate static pentru vector (elementele vectorului se reţin 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* şi *realloc*, ale căror definiții le găsim în fişierul 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 memoriei ce se doreşte a fi alocată nu este validă, adică este negativă).

Funcția *malloc* primeşte un parametru de tip întreg, care reprezintă numărul de octeți de memorie ce se doreşte 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 doreşte 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 (şi 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 zona 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* şi *realloc* se face cu ajutorul funcției *free*, care primeşte 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 şi 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* şi *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 aloca memorie necesară stocării celor *m* x *n* elemente ale matricii. În final vom face legăturile între vectorul de adrese şi zona de memorie unde vor fi stocate elementele matricii. În figura 1 este prezentată schema de alocare dinamică de memorie descrisă:

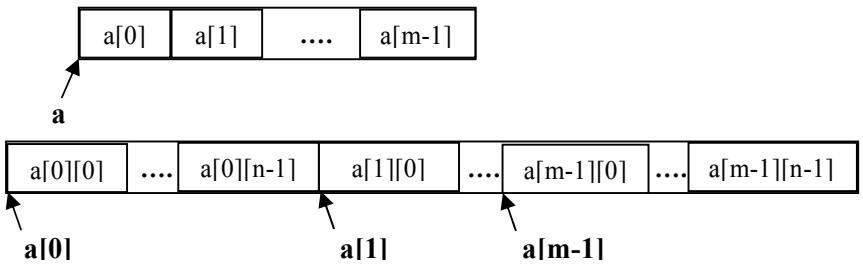


Fig. 1: Prima schemă de alocare dinamică a memoriei pentru o matrice

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* x *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]* ş.a.m.d., iar la sfârşit la adresa *a[n-1]* avem elementele ultimei linii.

Eliberarea zonei de memorie ocupate de matricea *a* se face în doi paşi: întâi se eliberează vectorul cu cele *m* x *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* x *n* x *sizeof(float)* şi 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 şi 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 şi 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* paşi: întâi se eliberează cele *m* zone de memorie în care sunt reținute elementele matricii şi 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* şi coloana *j* se face sub forma *a[i][j]*.

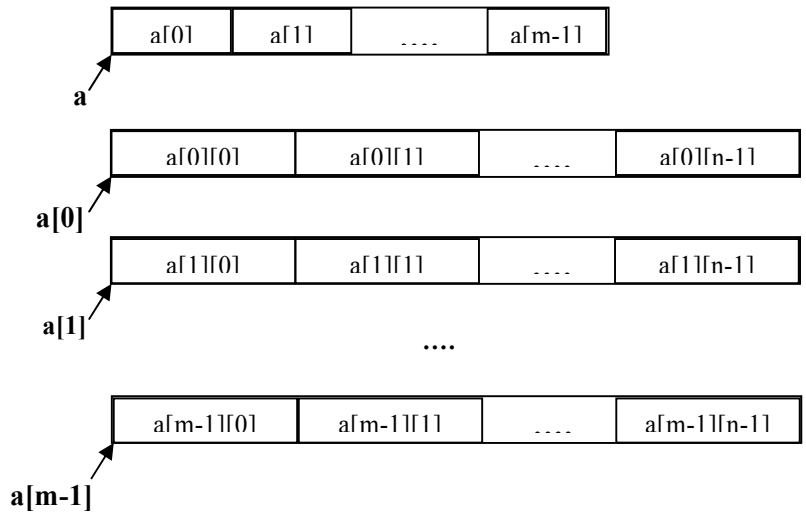


Fig. 2: A doua schemă de alocare a memoriei pentru o matrice

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 aloca dinamic memorie pentru matrici de ordin superior (3, 4 etc.). Lăsăm ca exercițiu cititorului scrierea codului C de alocare dinamică şi eliberare a memoriei pentru o matrice tridimensională.
2) Procedând în mod asemănător cu alocarea dinamică a unei matrici putem aloca 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** şi **realloc**, ale căror antete se găsesc în fişierul **malloc.h**. Eliberarea memoriei alocate dinamic în C se face cu ajutorul funcției **free**.

În C putem aloca memorie pentru un tablou (vector, matrice etc.) atât în mod static cât şi 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 şi 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 şi apoi eliberăm memoria ocupată de vectorul de adrese de început a liniilor matricei.

Teme

1. De la tastatură se citeşte 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 ş.a.m.d.). Să se citească de la tastatură elementele matricei, să se calculeze şi să se afişeze mediile aritmetice pe linii. În final se va elibera memoria alocată dinamic pentru stocarea matricei.
2. De la tastatură se citesc trei număre naturale nenule *m, n* şi *p*. Să se aloce dinamic memorie pentru o matrice tridimensională de dimensiuni *m, n* şi *p*. Să se citească de la tastatură elementele matricei. Să se găsească cel mai mare şi cel mai mic element al matricei. În final să se elibereze memoria alocată dinamic pentru stocarea matricei.
3. Construiți dinamic un vector în care să depuneți primele *n* numere naturale prime, unde *n* este citit de la tastatură.

8. Funcții în C

Obiective

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

În C, C++ şi Java (limbajele ce au la bază standardul ANSI C) ca subprograme nu avem decât funcții, o procedură putând fi implementă 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 şi ca o procedură (ignorându-se valoarea returnată). De exemplu, funcția *printf* returnează valoarea *int*, care reprezintă numărul de octeți care s-au afişat pe ecran. De cele mai multe ori (după cum am văzut şi în exemplele de mai sus) funcția *printf* este apelată ca o procedură, ignorându-se valoarea returnată.

După cum am mai spus şi în primul capitol, funcţiile pot fi descrise atât la începutul programului, deasupra funcţiei principale, cât şi după aceasta, situaţie în care definiţiile funcţiilor trebuie totuşi să apară deasupra funcţiei principale.

Structura unei funcţii în C este următoarea:

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

Definiţia funcţiei începe cu *tip*, care este tipul valorii pe care o returnează funcţia. Dacă *tip* nu este *void*, atunci funcţia va conţine de regulă cel puţin o instrucţiune *return expresie*. Dacă execuţia programului ajunge la o astfel de instrucţiune, atunci se evaluează expresia a cărei valoare trebuie să fie de tipul *tip* sau unul compatibil şi valoarea obţinută se returnează, execuţia programului continuând cu instrucţiunile de după locul în care a fost apelată funcţia.

Dacă o funcţie ce returnează o valoare de un tip diferit de *void* nu se termină cu o instrucţiune *return*, atunci la compilare vom obţine mesajul de atenţionare *Warning 5: Function should return a value*.

Spre exemplificare vom scrie o funcţie care returnează ca valoare a funcţiei 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 funcţiei *medie* după funcţia 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 şi virgulă.

În C parametrii ce se returnează din funcție (variabilele de ieşire) 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 transmişi prin valoare (aşa cum este cazul parametrilor *x* şi *y*), sunt copiate în zone de memorie noi şi de aceea, dacă le modificăm, la părăsirea funcției, valorile modificate se pierd, zonele de memorie alocate pentru aceşti 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* şi de aceea valoarea calculată rămâne în variabila *med* după ce se părăseşte 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 şi eventualele modificări în interiorul funcției asupra elementelor tabloului păstrându-se şi la părăsirea funcției.

În continuare prezentăm o funcție care calculează media aritmetică a elementelor unui şir 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 şi în variabila transmisă prin adresă, aşa 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 şi 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 şi se doreşte a fi returnată o singură adresă, din comoditate şi pentru a fi mai uşor de utilizat şi de înțeles, noua adresă se returnează ca valoare a funcției. Aşa este cazul funcțiilor de alocare dinamică de memorie din fişierul antet *alloc.h* (*malloc*, *calloc*, *realloc*). Fiecare dintre aceste funcții returnează adresa zonei de memorie nou alocate ca valoare a funcției. Rescriem şi 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 şi de lista de argumente dată între paranteze. Dacă o funcție are tipul returnat **void**, atunci ea nu returnează nici o valoare şi se apelează practic ca o procedură.

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

Teme

1. Scrieți o funcție pentru căutare binară a unei valori într-un şir de numere reale sortat crescător.
2. Scrieți o funcție pentru căutare secvențială a unei valori într-un şir de numere întregi. Funcția returnează numărul de apariții a unei valori întregi în şir.
3. Scrieți o funcție care sortează crescător un şir de numere reale.

9. String-uri

Obiective

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

În C, un şir de caractere care se termină cu caracterul *NULL* sau *'\0'* sau pur şi simplu *0* (de unde şi denumirea de şir 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 şir, alocarea memoriei pentru un *string* se poate face atât static cât şi dinamic.

În fişierul antet *stdio.h* avem definite funcțiile *gets* şi *puts* pentru citirea unui string de la tastatură, respectiv afişarea 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 întâlneşte caracterul spațiu sau unde s-a apăsat *Enter*. De exemplu, dacă se citeşte *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 afişează un string *s* cu *puts(s);* atunci după afişarea şirului de la adresa *s* se face salt la începutul liniei următoare, ceea ce nu se întâmplă dacă afişarea se face cu *printf("%s",s)*.

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

1) *strcat(s1, s2)* la sfârşitul string-ului *s1* se adaugă *s2* şi 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 şirurile sunt identice ca şi 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 acelaşi lucru ca şi *strcmp*, cu deosebirea că nu se face distincție între litere mari şi mici.
5) *strcnmp(s1,s2,n)* realizează o comparație între primele cel mult *n* caractere ale string-urilor *s1* şi *s2*.
6) *strcnmpi(s1,s2,n)* realizează o comparație între primele cel mult *n* caractere ale string-urilor *s1* şi *s2* fără a se face distincție între litere mari şi mici.
7) *strcpy(s1,s2)* copiază caracterele de la adresa *s2* (până se întâlneşte 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 sfârşitul 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 sfârşit 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)* înlocuieşte primele cel mult *n* caractere ale string-ului *s1* cu primele cel mult *n* caractere din string-ul *s2*.

Ca aplicaţii pentru string-uri propunem:

1. O funcţie care returnează printre parametri ei toate poziţiile pe care apare un string *s2* într-un string *s1*.
2. Funcţie pentru înlocuirea primei apariţii în *s1* a string-ului *s2* cu string-ul *s3*.
3. Funcţie pentru înlocuirea tuturor apariţiilor lui *s2* în *s1* cu string-ul *s3*.

Pentru aceste aplicaţii vom folosi funcţii din fişierul antet *string.h*, dar va trebui şi să lucrăm direct cu adrese de memorie şi 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 strnfind(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 strreplace1(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++;
		strreplace1(s1,s2,s3);	/* se inloc. s2 cu s3 in s1 */
		poz=strstr(s1,s2);	/* se cauta dinnou 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=strnfind(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("'%s' nu apare in '%s'\n",s2,s1);
        getch();
}
```

Rezumat

String-ul se memorează în C într-un şir de caractere obişnuit. Caracterul '\0' (caracterul cu codul ASCII 0, adică NULL) marchează sfârşitul string-ului. De aceea, despre string în C se spune că este un şir NULL terminat.

În fişierul antet **string.h** găsim definiţiile funcţiilor de lucru cu string-uri în C.

Teme

1. Folosind funcţia *strrev* scrieţi o funcţie 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 aceleaşi cu situaţia în care sunt citite de la dreapta spre stânga.
2. Scrieţi o funcţie care returnează numărul de cuvinte dintr-un string primit ca parametru. Cuvintele se consideră a fi grupuri de caractere despărţite prin spaţii.

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 aşa numitele câmpuri ale structurii. O structură se defineşte 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 şi virgulă.

Între acolada } ce încheie definirea structurii şi semnul ; (punct şi virgulă) putem declara variabile care vor fi evident 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 combinaţie 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 afişa:

```
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 defineşte cu ajutorul cuvântului rerervat **struct**. Accesul la membrul unei structuri se face ajutorul operatorului punct sub forma *variabila_tip_struct.membru*.

Temă

De la tastatură citiți stocul unei firme într-un şir de elemente de tip *struct*. În structură se vor reține: denumirea produsului, cantitatea şi prețul. Sortați crescător şirul după denumirea produselor, după care afişați 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|
-----------------------------------------------------------------
```

11. Pointeri către structuri. Liste înlănțuite.

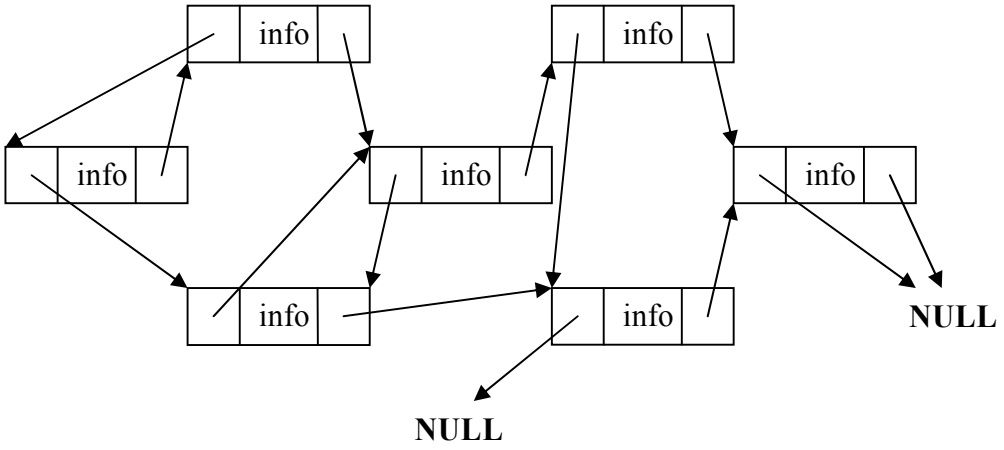
Obiective

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 şi 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 aşa numite noduri legate între ele. În fiecare nod se memorează nişte informații (numere, caractere, string-uri etc.) şi 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 şi 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 şi pentru fiecare nod *x* diferit de *cap* există un singur nod *y* care este legat de *x*.

După modul de introducere şi 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 ieşit. O stivă (după cum sugerează şi 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.

Coada este tot o listă liniară, dar de tipul FIFO (First In First Out), adică primul intrat – primul ieşit. După cum sugerează şi 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. Coada poate fi şi 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 reține în permanență adresa (într-o variabilă de tip pointer denumită de obicei *ultim*) ultimului nod introdus în listă şi 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 numeşte coadă circulară.

Cu ajutorul pointerilor către structuri putem implementa în C listele înlănțuite. Un pointer către o structură se defineşte 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* şi *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) /* scoatera 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 –> 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 şirurile *stg*, *dr*, şi *info*. Vectorii *stg* şi *dr* memorează indicii nodului fiu stâng, respectiv fiu drept, iar în *info* se reţin informaţiile din fiecare nod.
3) Afişarea informaţiei reţinute în frunzele unui arbore binar.
4) Afişarea informaţiei reţinute pe un anumit nivel al unui arbore binar.
5) Căutarea unei informaţii într-un arbore binar.

12. Fişiere în C

Obiective

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

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

12.1. Funcţii de lucru cu fişiere

Funcţiile de lucru cu fluxuri în C sunt definite în *stdio.h*. Aşa că atunci când lucrăm cu fişiere în C trebuie să includem acest fişier antet.

Deschiderea fişierului se face folosind funcţia *fopen*, care are următoarea definiţie (antet):

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

Funcţia primeşte ca argumente două string-uri, primul trebuie să conţină numele fişierului (precedat eventual de cale) şi al doilea modul de deschidere al fişierului. Se returnează pointerul către fluxul deschis (pointer către *FILE*). Dacă nu s-a putut deschide fişierul, se returnează adresa nulă (*NULL*).

Deschiderea unui fişier pentru operaţii de scriere eşuează dacă numele lui (inclusiv calea) conţine caractere invalide, fişierul este deja deschis de o altă aplicaţie sau dacă nu avem drepturi de scriere în calea respectivă.

Un fişier nu poate fi deschis pentru a se citi din el informaţii decât dacă fişierul există, nu are atributul *hidden* (ascuns) şi avem drepturi de citire asupra lui.

Putem deschide un fişier pentru scriere, pentru citire sau pentru adăugare la sfârşit (*append*), în modul binar (fişierul în acest caz e interpretat ca o succesiune de octeţi, baiţi) sau în modul text (fişierul e interpretat ca o succesiune de linii cu texte despărţite prin caracterele *'\r'* şi *'\n'*).

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

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

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

Iată câteva exemple de moduri de deschidere:

"rb" – Fişierul se deschide numai pentru citire în modul binar
"wb" – Fişierul se deschide numai pentru operaţii de scriere în modul binar
"ab" – Fişierul este deschis pentru adăugare la sfârşit în modul binar
"rb+" – Fişierul este deschis pentru citire în modul binar, dar sunt suportate şi operaţii de scriere
"wb+"– Fişierul este deschis pentru scriere şi citire în modul binar
"ab+" – fişierul este deschis pentru scriere cu adăugare la sfârşit în modul binar, dar se pot face şi citiri din fişier.

Dacă mai sus nu apărea caracterul *'b'* sau apărea caracterul *'t'*, atunci obţineam modurile similare text de deschidere de fişier.

Facem observaţia că semnul plus + poate fi pus şi între cele două litere. De exemplu în loc de **"ab+"** putem pune **"a+b"**.

În exemplul următor deschidem un fişier pentru operaţii de citire, dar şi 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 fişierului se face cu ajutorul funcţiei *fclose*:

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

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

Pentru a închide toate fişierele deschise la un moment dat putem folosi funcţia:

```
int fcloseall();
```

Funcţia returnează numărul de fişiere ce au fost închise.

Verificarea dacă s-a ajuns la sfârşitul fişierului se face cu ajutorul funcţiei:

```
int foef(fis);
```

Funcţia returnează *0* dacă nu s-a ajuns la sfârşitul fişierului sau o valoare nenulă dacă este sfârşit de fişier.

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

```
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 fişierelor deschise se face cu:

```
int flushall();
```

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

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

Citirea unui caracter din fişier (şi se returnează codul caracterului citit sau *EOF* dacă s-a ajuns la sfârşitul fişierului) se face cu:

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

Pentru a scrie un caracter în fişier folosim funcția:

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

Citirea unui şir de caractere se face cu:

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

Pentru această funcție, fişierul este interpretat ca fiind text. Se citeşte un şir de caractere de lungime maximă *n*, şir care se depune la adresa *s*. Lungimea şirului 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 sfârşitul liniei. La sfârşitul şirului citit se depune caracterul '\0'.

Scrierea unui string într-un fişier text (la sfârşitul string-ului în fişier se pun caracterele cu codurile 10 şi 13, adică este vorba de carriage-return *'\r'* şi respectiv new-line *'\n'*):

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

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

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

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

Citire formatată în modul text (varianta *scanf* pentru fişiere 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 fişierul indicat de pointerul *fis* şi se depun la adresa de memorie *p*. Funcţia returnează numărul efectiv de blocuri citite din fişier (*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* şi se scriu în fişierul *fis*. Funcţia returnează numărul efectiv de blocuri scrise în fişier.

Există evident şi o funcţie pentru salt în fişier (mutarea poziţiei curente):

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

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

SEEK_SET sau 0 - de la începutul fişierului
SEEK_CUR sau 1 - de la poziţia curentă
SEEK_END sau 2 - de la sfârşitul fişierului.

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

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

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

Pentru returnarea poziţiei curente (în octeţi de la începutul fişierului) vom folosi funcţia:

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

Folosind funcţia *ftell* putem scrie o altă funcţie care să returneze lungimea fişierului:

```
# 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("Fisierul 'autoexec.bat' are %ld octeti.\n",
		flung("c:\\autoexec.bat"));
	getch();
}
```

Redenumirea unui fişier se face cu ajutorul funcției:

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

Fişierul cu numele *numef1* se redenumeşte cu numele *numef2*. Numele de fişiere din string-urile *numef1* şi *numef2* pot fi precedate şi de cale. Redenumirea nu se poate face decât pe acelaşi drive. De exemplu, apelul funcției *rename("c:\\fisier1.txt", "a:\\fisier2.txt")* se soldează cu eşec (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* şi variabila globală *errno* se setează la una dintre valorile:

ENOENT – *No such file or directory* (nu există fişierul *numef1*, sau *numef2* este un nume de fişier 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 fişiere indică drive-uri diferite).

Ştergerea unui fişier de pe disc se face cu:

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

Se încearcă ştergerea fişierului 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* şi variabila globală *errno* se setează la una dintre valorile:

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

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

1) *Concatenarea mai multor fişiere ale căror nume sunt primite în linie de comandă. Ultimul parametru primit în linie de comandă se consideră a fi numele fişierului 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("Fisierele se depun in <fis_destinatie>.\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);
	cprintf("Am afisat: %ld linii.",n);
	getch();
	return 0;
}
```

3) Produsul a două matrici citite din fişiere 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 fişierelor pe disc în DOS ne folosim de funcţiile:

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

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

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

În parametrul *cale* se transmite calea unde se face căutarea şi structura numelor de fişier căutate. De exemplu, dacă parametrul *cale* conține *"c:\\Windows*.bmp"*, atunci se vor căuta fişierele 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 fişierul găsit: numele său, dimensiunea (în octeți), data şi ora creării, atributul fişierului (*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 fişierelor 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 afişează toate fişierele cu extensia *txt* din directorul curent:

```
void dirtxt(void)
{
     struct ffblk ff;
     int gata;
     puts("Fisierele *.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 fişier în C trebuie să definim o variabilă de tip pointer către structura **FILE**. Deschiderea fişierului se face apelând funcţia **fopen**. Funcţia **fopen** primeşte ca parametri numele fişierului şi modul în care vrem să deschidem fişierul. Dacă fişierul este deschis cu succes. Funcţia **fopen** returnează pointerul către tipul FILE cu ajutorul căruia se va prelucra fişierul. Un fişier deschis cu fopen poate fi închis folosind funcţia **fclose**. Scrierea/citirea în/din fişier în modul text se poate face cu **fprint**, respectiv cu **fscanf**. Pentru modul binar folosim funcţiile **fwrite** şi **fread**. Poziţionarea în fişier se face cu **fseek**.

Teme

1. Scrieţi o funcţie care returnează numărul de apariţii a unui cuvânt într-un fişier text. Funcţia primeşte ca parametrii cuvântul şi numele fişierului.
2. Dintr-un fişier text se citesc elementele unui şir de numere reale. Să se sorteze crescător şirul şi să se depună după aceea vectorul în acelaşi fişier.

13. Variabile statice

Obiective

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

În C, o variabilă locală într-o funcţie poate fi declarată ca fiind statică. Pentru această variabilă se păstreză valoarea şi, când se reapelează funcţia, variabila va fi iniţializată 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 sfârşitul execuţiei programului, dar accesul la variabilă nu este posibil decât din interioul funcţiei 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 iniţializată la începutul execuţiei programului cu valoarea 1. La fiecare apel al funcţiei *fct* variabila *x* se incrementeză cu o unitate şi se afişează noua sa valoare. Aşadar pe ecran în urma execuţiei 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.

14. 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 aceeaşi categorie) funcțiile cu listă variabilă de parametri nu funcționează corect pentru tipurile de dată *char*, *unsigned char* şi *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 ieşirea 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 neaşteptate. 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 transmiteam toți cei cinci ultimi parametrii 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 şi tipul parametrilor de apel.

În final trebuie să facem observația că limbajul C++ oferă câteva alternative mai simplu de implementat şi 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, aşa cum este cazul funcțiilor **printf** sau **scanf**. Aceste funcții se implementează cu ajutorul tipului **va_list** şi a macrocomenzilor **va_start** şi **va_end**, toate definite în fişierul 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 şi minimul unor valori reale.

ANEXĂ - Urmărirea execuției unui program. Rulare pas cu pas.

Pentru a vedea efectiv traseul de execuție şi modul în care se îşi 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 şi 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++ şi 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 afişează 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 afişării mediului de programare. Reafişarea mediului de programare se poate face cu *Repaint desktop* din meniul ≡. Este bine de ştiut 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. Aşa că programul, după ce a fost depanat şi este terminat, este indicat să fie compilat şi 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++ şi **F9** în Visual C++. Linia va fi marcată (de obicei cu roşu). 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++ şi **Shift+F5** în Visual C++.

Dacă se doreşte continuarea execuției programului fără *Debbuger*, se poate apăsa **Ctrl+F9** în Borland C/C++ şi **F5** în Visual C++.

În orice moment, în Borland C/C++ de sub DOS rezultatele afişate 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 şi C++*, Editura Univ. "Transilvania", Braşov, 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 Annotatted C++ Referense 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. Ewards, 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.