CUPRINS

CUPRINS ____ *i*

Cap 1 INTRODUCERE ÎN LIMBAJUL C ____ *4*

1.1 Scurt istoric ____ **4**

1.2 Forma unui program C ____ **8**

1.3 Compilarea unui program C ____ **9**

Cap 2 Tipuri, operatori, expresii ____ *11*

2.1. Tipuri simple ____ **11**

2.2. Literali ____ **12**

2.3. Declararea unei variabile şi a unei constante ____ **13**

2.4. Operatori ____ **14**

2.4.1. Operatori aritmetici ____ 15
2.4.2. Operatorii relaţionali (de comparaţie) ____ 15
2.4.3. Operatori logici ____ 16
2.4.4. Operatori la nivel de bit ____ 16
2.4.5. Operatorul de asignare ____ 16
2.4.6. Operatorul condiţional ____ 17
2.4.7. Operatorul virgulă ____ 17
2.4.8. Precedenţa operatorilor ____ 18

Cap 3 FUNCTII ____ *19*

3.1. Definirea unei funcţii ____ **19**

3.2. Returnarea unui apel ____ **20**

3.3. Funcţii cu un număr variabil de parametri ____ **20**

3.4. Sfârşitul execuţiei unui program ____ **20**

3.5. Apelul şi transmiterea parametrilor ____ **21**

3.6. Funcţia principală main ____ **22**

Cap 4 FUNCŢII DE INTRARE IEŞIRE ____ *24*

4.1.Fluxuri şi fişiere ____ **24**

Semnificaţia ____ **26**

4.2. Funcţii de intrare/ieşire pentru caractere ____ **27**

4.3. Scrierea cu format ____ **28**

4.5. Citirea cu format ____ **32**

Cap 5 INSTRUCŢIUNI DE CONTROL ____ *35*

5.1. Instrucţiunea if ____ **36**

5.3. Instrucţiunea while ____ **38**

5.4. Instruc]iunea do ... while ____ **39**

5.5. Instrucțiunea for 39
5.6. Instrucțiunea break 41
5.7. Instrucțiunea continue 42
5.8. Instrucțiunea go to 42
5.9. Exerciții 43
1. Calculul factorialului. 43
2. Conversie 43
3. Numărarea biților 1 într-un număr 43
4. Prezentare de date 43
5.10. Soluții la exerciții 43
1. Calculul factorialului 43
2. Conversie 44
3. Numărarea biților 1 într-un număr 45
4. Prezentare de date 45
Cap 6 TABLOURI ŞI POINTERI *47*
6.1. Pointeri 47
6.2. Tablouri cu o dimensiune 48
6.2.1. Referențierea unui element al tabloului 48
6.2.2. Inițializarea unui tablou 48
6.3. Relația între pointeri şi tablouri 50
6.4. Operații aritmetice cu pointeri 51
6.5. Şiruri de caractere 51
6.6. Tablouri cu mai multe dimensiuni 51
6.7. Tablouri de pointeri 53
6.8. Pointeri şi alocarea memoriei 54
6.9. Operatorul "sizeof" 55
6.10. Pointeri către funcții 55
6.11. Exerciții 57
6.12. Soluții 57
Cap 7 CONSTRUIREA DE NOI TIPURI *59*
7.1. Redenumirea unui tip 59
7.2. Tipul structură 60
7.3. Accesul şi inițializarea câmpurilor unei structuri 61
7.4. Structuri autoreferențiate 61
7.5. Tipul enumerare 63
7.6. Uniuni 63
7.7. Exerciții 65
7.8. Soluții 66
Cap 8 GESTIUNEA MEMORIEI *70*

8.1. Precizarea unui mecanism de memorare 71
8.2. Gestiunea automată a memoriei 71
8.3. Gestionarea "register" a memoriei 71
8.4. Gestionarea statică a memoriei 72
8.5. Variabile globale 72
8.6. Declararea variabilelor externe 72
8.7. Gestiunea dinamică a memoriei 73
8.8. Exerciții 73
8.9. Soluții 74
Cap 9 CONVERSIA DE TIP 76
9.1. Mecanisme de conversie 76
9.1.1 Conversia la asignare 77
9.1.2 Evaluarea expresiilor aritmetice 77
9.1.3. Evaluarea expresiilor logice 78
9.1.4 Conversii explicite 78
9.1.5 Conversie de pointeri. Pointerul universal 79
Cap 10 PREPROCESORUL 80
10.1 Directive preprocesor 80
10.1.1 Constante simbolice. Directiva #define 81
10.1.2 Includerea fişierelor 81
10.1.3 Compilare condiționată 82
10.1.4 Directiva `#error` 82
10.2 Macroinstrucțiuni 83
10.2.1 Macroinstrucțiuni predefinite 83
10.2.2 Macroinstrucțiuni tip funcție 83
Cap 11 EXERCIȚII 84
BIBLIOGRAFIE 86
ANEXA 87
Biblioteca standard C 87
Codul ASCII 90

Cap 1 INTRODUCERE ÎN LIMBAJUL C

1.1 Scurt istoric
1.2 Forma unui program C
1.3 Compilarea unui program C

1.1 Scurt istoric

Strămoşii limbajului C sunt limbajele de programare CPL, BCPL, B şi Algol 68. CPL a fost dezvoltat la Universitățile Cambridge şi London în anii 1960-1963. BCPL a fost proiectat de Martin Richards în 1967 şi face parte din categoria "low-level languages", sau "systems programming languages". În anii '60, la Bell Laboratories în USA, Ken Thomson a început proiectarea sistemului de operare Unix. Primul limbaj de nivel înalt implementat sub acest sistem de operare a fost limbajul B, un limbaj bazat pe BCPL care a fost proiectat de asemenea de către Ken Thompson în 1970. Asemănarea cu BCPL a fost mai degrabă semantică; sintaxa este total diferită. Proiectarea limbajului B a fost influiențată de limitele maşinii pe care a fost implementat: un PDP-7 cu capacitatea 4K (cuvinte de 18 biți).

Limbajele BCPL şi B sunt limbaje de nivel scăzut față de limbajul Pascal şi nici unul din ele nu este tipizat: toate datele sunt considerate cuvinte maşină încât, exemple simple de programe duc la complicații nebănuite. În una din implementările BCPL-ului, operanzii flotanți din expresii trebuiau precedați de punct; restul operanzilor erau considerați întregi.

Problemele create în jurul limbajelor BCPL şi B au dus la dezvoltarea unui nou limbaj, bazat pe B, care la început s-a numit NB dar numele său a fost stabilit C. Numele "**C**", considerat la început a proveni de la **C**ambridge, este oficial din "**C**ombined" dar neoficial se consideră a fi inițiala pronumelui lui

Cristopher Strachey, cel care a condus echipa ce a proiectat CPL-ul. Limbajul a fost proiectat şi implementat de către Dennis Ritchie la Bell Laboratories în 1972. În mare parte (construcţiile for şi switch, tratarea pointerilor etc.) limbajul a fost influienţat şi de Algol 68. Genealogia limbajului C este prezentată în Figura 1.1.

Ritchie a lucrat pe un calculator DEC PDP-11 sub sistemul de operare Unix. Din acest motiv varianta clasică a limbajului C a fost asociată sistemului de operare Unix - versiunea 5 şi multă vreme a constituit un standard (neoficial de altfel) al limbajului [KeR 78].

Odată cu creştere popularităţii calculatoarelor personale, au apărut numeroase alte implementări ale limbajului C şi astfel a crescut şi popularitatea acestuia. Un fapt cu totul inedit a fost acela că, mare parte din implementările C -ului s-au dovedit de un mare grad de compatibilitate: un program scris pentru una din variante putea fi compilat cu succes folosind o altă variantă. Se înţelege că au existat şi neconcordanţe în multitudinea de "dialecte" ale limbajului. Pentru eliminarea acestora în 1983 a fost creată o comisie care să stabilească un standard al acestui limbaj. Standardul ANSI (American National Standard Institute) a fost redactat abia în decembrie 1989 şi a fost pus la dispoziţia celor interesaţi la începutul lui 1990. Firma Borland a folosit acest standard în faza de proiect şi a realizat implementările Turbo C care se bucură de un succes remarcabil. De altfel, în prezent , toate compilatoarele C sunt compatibile cu standardul ANSI C; unele din acestea au adăugate o serie de facilităţi sau extensii proprii care nu împiedică compilarea unui program standard.

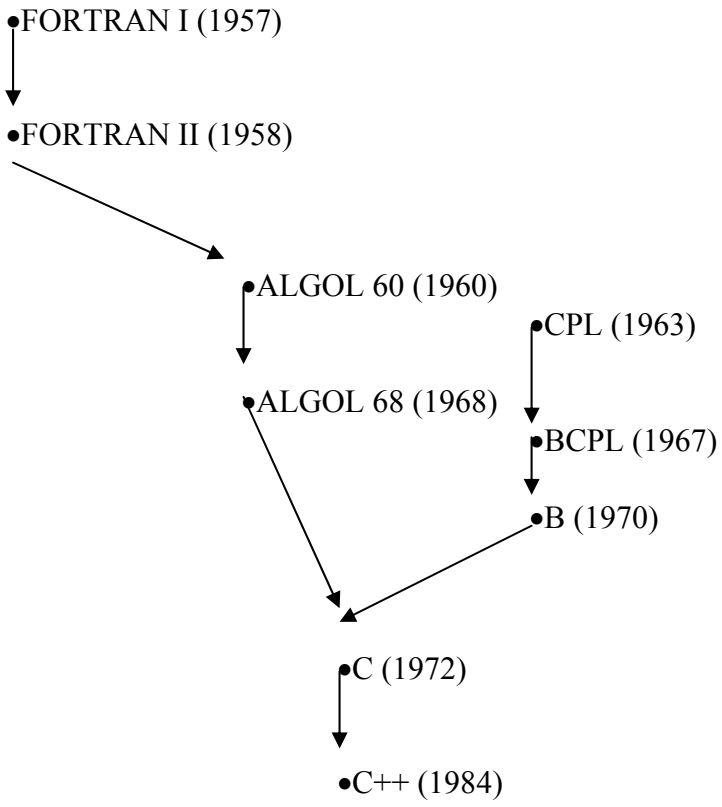


Figura 1.1.

Limbajul C este privit ca un limbaj de nivel mediu întrucât combină cele mai reuşite elemente ale limbajelor de nivel înalt (Pas-cal, Fortran, Algol, Modula, Ada) cu gradul de control şi flexibilitatea oferite de către limbajul de asamblare. C permite manevrarea biţilor, a octeţilor şi a adreselor de memorie, adică a elementelor de bază în funcţionarea unui calculator. În ciuda acestui fapt , programele scrise în C

au un grad mare de portabilitate: uşurința de a adapta programele scrise pentru o anumită categorie de calculatoare sau sisteme de operare la alte calculatoare sau sisteme de operare.

Limbajul C are structuri de control adecvate precum şi facilități de structurare a datelor care îl fac să fie folosit într-o diversitate de domenii. Are de asemenea o mulțime bogată de operatori care-i dau un înalt grad de expresivitate. Unul din motivele pentru care limbajul C este în acelaşi timp agreat de mare parte din programatori şi dezagreat de altă mare parte din programatori, este lipsa totală a verificării tipului (type checking). De exemplu, o variabilă de un tip scalar oarecare poate apare în aceeaşi expresie cu o variabilă de un alt tip scalar. Cei care agrează limbajul C îi apreciază flexibilitatea sa, cei care nu-l agrează îl consideră un limbaj lipsit de siguranță.

Ceea ce este sigur, este că datorită faptului că face parte din sistemul de operare Unix, limbajul a căpătat o mare popularitate în lumea academică şi nu numai. Compilatoarele oferite împreună cu sistemul Unix sunt relativ ieftine (uneori chiar gratuite) şi pot fi folosite pe o diversitate de maşini.

O nouă versiune a limbajului C, numită C++, a apărut în 1984. Originea acestui limbaj şSTR 94ț trebuie căutată în proiectul care a început în Aprilie 1979 la "Computing Research Center of Bell Laboratories" în Murray Hill, New Jersey, USA, proiect ce trebuia să realizeze distribuirea nucleului UNIX într-o rețea de computere conectate LAN. Pentru realizarea acestui proiect, Bjarne Stroustrup a realizat în Octombrie 1979 un preprocesor, numit Cpre, care a adăugat la limbajul C conceptul de clasă folosit pentru prima dată de limbajul Simula (dezvoltat în anii 1962-1967 de norvegienii Kristen Nyagaard şi Ole-Johan Dahl). În martie 1980 preprocesorul îmbunătățit era implementat pe 16 sisteme iar limbajul acceptat de acesta a fost numit "C cu Clase". Motivele pentru care a fost ales limbajul C (în fața altor limbaje ale vremii: Modula-2, Ada, Smalltalk, Mesa, Clu) pentru acest proiect sunt expuse chiar de creatorul noului limbaj:

- *C este un limbaj flexibil*: se poate folosi în orice domeniu şi permite utilizarea oricăror tehnici de programare;
- *C este un limbaj eficient*: semantica sa este "low level" ceea ce înseamnă că pot fi folosite cu eficiență resursele hardware pentru programele C. Aşa cum vom vedea pe parcursul prezentării limbajului, acesta oglindeşte conceptele fundamentale ale unui computer tradițional;
- *C este un limbaj disponibil:* un limbaj pentru care se poate găsi cu uşurință un compilator, indiferent de ce calculator dispunem;
- *C este un limbaj portabil*: chiar dacă un program C nu este automat portabil de la o maşină la alta, acest lucru este posibil în general fără un efort deosebit.

Cele mai mari influiențe asupra noului limbaj le-au avut: Simula care a dat clasele, Algol 68 care a dat supraîncărcarea operatorilor, referințele şi abilitatea de a declara variabile oriunde în cadrul unui bloc şi BCPL de unde a fost luată forma de prezentare a comentariilor cu //. După numele " C cu clase" limbajul a fost "botezat" C84. Asta pentru scurt timp însă deoarece comitetul ANSI pentru standardizarea C-ului a atras atenția că se poate crea o confuzie cu această denumire; era vorba totuşi de un nou limbaj, chiar dacă era clădit pe C. Numele C++ a fost sugerat de Rick Mascitti, un membru al echipei lui Stroustrup, şi a fost folosit pentru prima dată în Decembrie 1983.Iată ce spune Stroustrup despre această alegere: *<<I picked C++ because it wos short, had a nice interpretation, and wasn't of the form "adjective C". In C++, ++ can, depending on context, be read as "next", "succesor" or "increment" , tough it is always pronounced "plus plus".>>*

În figura 1.2. este prezentată genealogia limbajului C++ iar figura 1.3. prezintă etapele importante din proiectarea limbajului.

Datele care apar în paranteză se referă la prima implementare disponibilă pentru limbajul respectiv(după [STR 94]). C++arm reprezintă "The Annotated C++ Reference Manual" iar C++std este standardul ANSI C++.

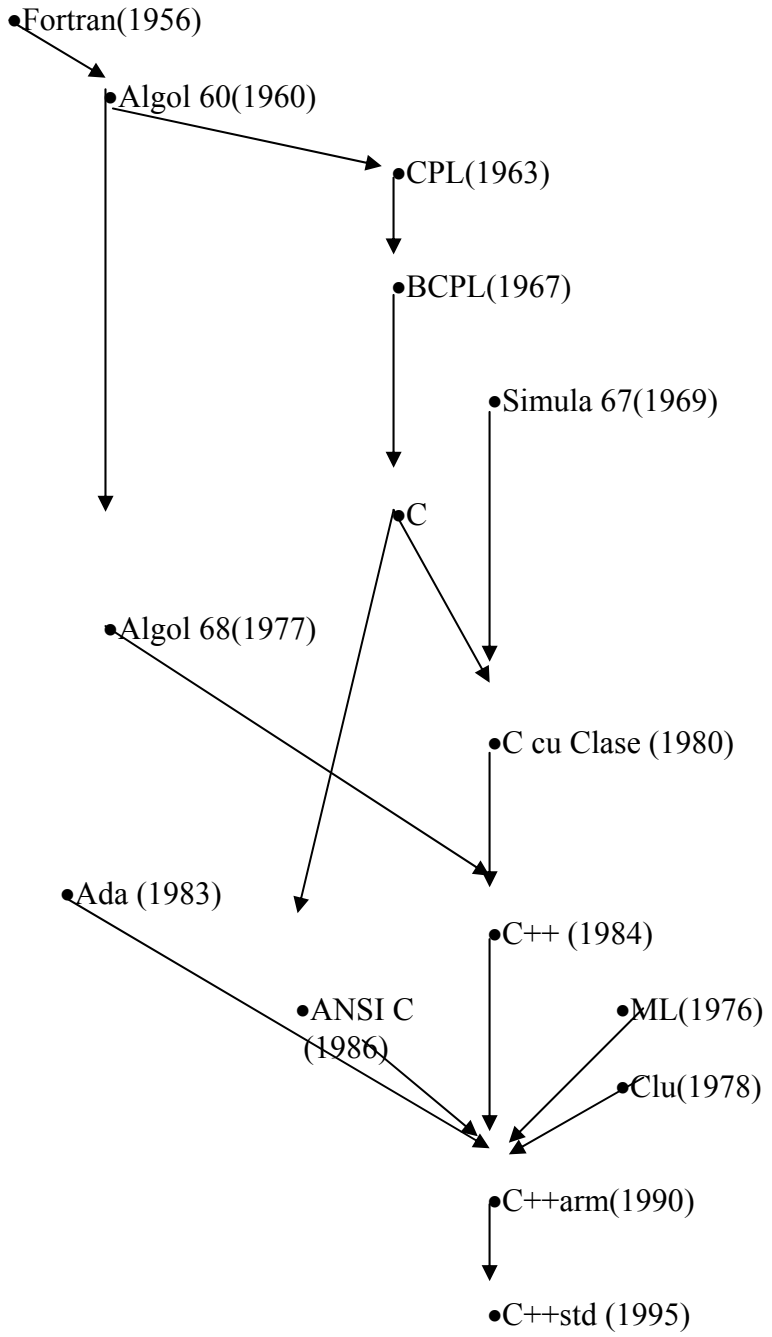


Figura 1.2.

1979	Mai	Începutul proiectului “C cu clase”
	Octombrie	Prima implementare a lui C cu clase
1980	Aprilie	Primul raport intern la Bell Labs asupra lui C cu clase
1982	Ianuarie	Primul articol extern asupra lui C cu clase
1983	August	Prima implementare a limbajului C++
	Decembrie	Limbajul a fost “botezat” C++
1984	Ianuarie	Primul manual C++
1985	Februarie	Prima lansare externă C++
	Octombrie	Manualul *The C++ Programming Language*
1986	Septembrie	Prima conferinţă OOPSLA (centrată pe Smalltalk)
1987	Noiembrie	Prima conferinţă USENIX C++

	Decembrie	Prima lansare GNU C++
1988	Ianuarie	Prima implementare Oregon Software C++
	Iunie	Prima implementare Zortech C++
1990	Martie	Prima reuniune tehnică ANSI X3J16 pentru C++
	Mai	Prima implementare Borland C++
	Mai	C++arm: *The Annotated C++ Reference Manual*
1991	Iunie	Prima reuniune ISO WG21
1992	Februarie	Prima lansare DEC C++
	Martie	Prima lansare Microsoft C++
	Mai	Prima lansare IBM C++
1994	August	A fost înregistrat Raportul Comitetului ANSI/ISO

Figura 1.3.

1.2 Forma unui program C

În general, cel care se apucă de învățat limbajul C, mai ştie cel puțin un limbaj de programare (Pascal în cele mai multe cazuri). Aceasta pentru că în programa de liceu, la disciplina de informatică, se începe cu un limbaj diferit de C iar în învățământul superior, la facultățile care au în planul de învățământ informatică şi chiar la cele de profil, primul limbaj de programare studiat (cel puțin până acum) a fost altul decât limbajul C. Vom începe prezentarea printr-un exemplu.

Programul 1

```
/* Calculul celui mai mare divizor comun (cmmdc) a doua numere intregi
   pozitive a si b. Metoda care se aplica pentru a determina cmmdc este
   urmatoarea:
   -    se calculeaza restul impartirii intregi a lui a prin b;
   -    daca acest rest este nul, b este valoarea cautata. In caz
      contrar, a capata valoarea lui b, b capata valoarea restului si
      procedeul se repeta.
*/
# include <stdio.h>
int cmmdc (int x, int y){
  int rest;
  do {
     rest = x%y;
     x = y;
     y = rest;
  }  while (rest!=0);
     return x;
}
  int main (){
    int a,b;
    char raspuns;
    do {
     printf ("  Introduceti 2 valori intregi pozitive -->");
     scanf ("%d%d", &a, &b);
     if (b>a) {
          int c = a;a = b;b = c;
     }
     printf (" Cmmdc al celor doua numere este:
                %d\n",cmmdc (a,b));
     printf (" Continuati ? d)a n)u -->");
```

```
    scanf ("%c", &raspuns);
   } while (raspuns!='n');
   return 0;
}
```

Să trecem în revistă structura programului prezentat.

Primele linii din program constituie comentarii. Aceste comentarii sunt incluse între "parantezele" "/*" şi "*/". Urmează o linie ce începe cu #. Pe această linie este o directivă care este gestionată de precompilator (preprocesor) şi permite includerea definiţiilor funcţiilor intrare/ieşire furnizate de sistem şi care se găsesc în fişierul **stdio.h**.

Programul pe care l-am prezentat este constituit din două funcţii: main şi cmmdc. Trebuie reţinut faptul că orice program C conţine definiţia unei funcţii cu numele main care ţine locul programului principal: punctul de intrare în execuţia unui program C este funcţia main. Corpul unei funcţii este delimitat de acolade: '{' ţine locul lui begin (dinalte limbaje) iar '}' al lui end.

În corpul funcţiei main în Programul 1 avem apeluri la alte funcţii: printf, scanf, cmmdc. În limbajul C nu există instrucţiuni de intrare/ieşire; operaţiile de intrare/ieşire se realizează prin apel la funcţiile unei biblioteci standard furnizate de sistem. Aceste funcţii vor fi prezentate mai târziu într-un capitol special. În exemplul de aici avem:

printf cu un parametru aflat între ghilimele: acest text va fi tipărit pe ecran

scanf cu trei parametri: "%d%d", &a şi %b. Acesta este un apel la o operaţie de citire a unor date pe care le introducem de la tastatură. Parametrul "%d%d" specifică formatul de citire a două numere întregi (d de la digit) iar parametrii &a, &b corespund adreselor (faptul că & apare în faţă) unde vor fi memorate cele două valori întregi: &a adresa variabilei a, iar &b adresa variabilei b.

printf ("Cmmdc al celor doua numere este: %dăn" cmmdc (a,b))

este un apel la o operaţie de scriere (pe ecranul monitorului) a textului: cuprins între " şi " în afară de %dăn care reprezintă formatul cu care se va tipări valoarea returnată de apelul la funcţia cmmdc (a,b): va fi scris un număr întreg (%d) după care se trece la linie nouă (caracterul ăn).

Definiţiile acestor funcţii se află în fişierul sistemului stdio.h.

Construcţia "do {..} while (..)" care apare şi în funcţia main şi în funcţia cmmdc este una din structurile iterative de care dispune limbajul C: grupul de instrucţiuni cuprins între acoladele de după do se execută atâta timp cât condiţia scrisă după while este îndeplinită (de precizat că acel grup de instrucţiuni se execută măcar odată).

Structurile de control ale limbajului C se vor prezenta într-un capitol special.

1.3 Compilarea unui program C

Se ştie că există două metode generale de transformare a unui program sursă într-un program executabil: compilarea şi interpretarea. Limbajul C a fost optimizat special în vederea compilării. Deşi au fost concepute şi interpretoare de C şi acestea sunt disponibile pentru diferite medii, C a fost gândit de la bun început ca fiind destinat compilării.

Crearea unui format executabil al unui program C parcurge următoarele trei etape:

1. Crearea programului;
2. Compilarea programului;
3. Legarea programului cu funcţiile necesare existente în bibliotecă.

Majoritatea compilatoarelor C asigură medii de programare care includ facilităţi pentru realizarea tuturor etapelor precizate mai sus. Dacă se foloseşte un compilator care nu are integrat un editor (cum sunt cele din sistemul de operare UNIX), atunci trebuie folosit un editor pentru crearea fişierului sursă C. Atenţie însă: compilatoarele acceptă ca fişiere de intrare (programe sursă) numai fişiere text standard. Prin urmare, dacă se crează un fişier sursă cu editorul Word su Windows, acesta trebuie salvat ca fişier text pentru a putea fi utilizat de un compilator.

Toate programele în C constau dintr-una sau mai multe funcţii; singura funcţie care trebuie să fie prezentă în orice program se numeşte **main( ).** Aceasta este prima funcţie apelată la începutul executării programului. Într-un program C corect conceput, funcţia main( ) conţine, în esenţă, o schemă a ceea ce face programul, schemă compusă din apeluri ale altor funcţii aflate în *biblioteca standard a limbajului C* sau definite de programator. Standardul ANSI C precizează setul minimal de funcţii care va fi inclus în biblioteca standard; compilatoarele ce există pe piaţă conţin cu siguranţă şi alte funcţii. Procesul de legare (linking) este acela în care programul de legare(linker) combină codul scris de programator cu codul obiect al funcţiilor utilizate şi care se găseşte în bibliteca standard. Unele compilatoare au programul propriu de legare, altele folosesc programul de legare al sistemului de operare.

Un program C compilat creează şi foloseşte patru zone de memorie disjuncte, care îndeplinesc funcţii distincte (figura 1.4.).

Prima regiune este aceea care stochează codul programului.Urmează o zonă de memorie în care sunt stocate variabilele globale. Celelalte două zone reprezintă aşa zisa stivă a programului şi zona de manevră. Stiva este folosită în timpul execuţiei programului în scopurile:

- memorează adresele de revenire ale apelurilor la funcţii;
- memorează argumentele funcţiilor;
- memorează variabilele locale;
- memorează starea curentă a unităşii centrale de prelucrare.

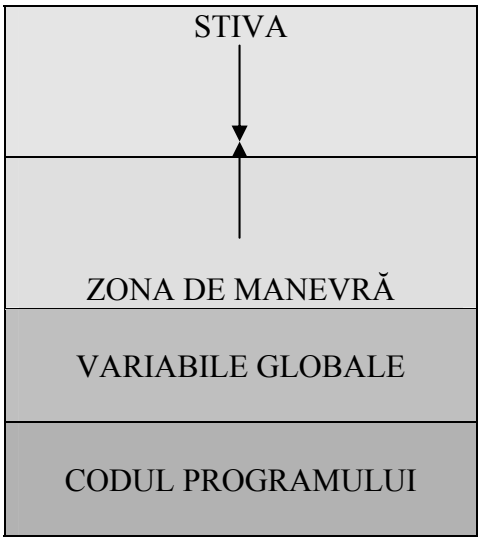


Figura 1.4.

Zona de manevră (heap) este o regiune de memorie liberă pe care programul , prin intermediul funcţiilor de alocare dinamică ale limbajului C, le foloseşte pentru stocarea unor articole de genul listelor înlănţuite şi arborilor.

Să mai precizăm , în încheierea acestei scurte introduceri , că în general, compilatoarele existente pe piaţă, sunt compilatoare C++. Aşa cum vom vedea în partea a doua a acestei cărţi, C++ este o versiune extinsă a limbajului C, care a fost concepută pentru programarea orientată pe obiecte. De aceea C++ acceptă limbajul C: orice program C va putea fi compilat de un compilator C++.După unii autori, programarea în limbajul C va dăinui multă vreme de acum încolo, aşa cum dăinuie de când a fost creat limbajul şi, un programator care nu ştie C nu va putea programa în C++.

Cap 2 Tipuri, operatori, expresii

2.1. Tipuri simple
2.2. Literali
2.3. Declararea unei variabile
2.4. Operatori

O expresie în limbajul C – ca în orice limbaj de programare – este o secvență de operatori şi operanzi care descrie algoritmul de calcul al unei valori. Operanzii într-o expresie sunt nume de variabile, nume de constante (literali), apeluri la funcții. Aceşti operanzi, fiecare dintre ei, au un anume tip iar în funcție de aceste tipuri se vor aplica algoritmi specifici pentru determinarea valorii expresiei.

2.1. Tipuri simple

Tipurile simple (sau tipurile de bază) ale limbajului C definesc dimensiunea zonei de memorie ocupată de valoarea unei variabile precum şi modul în care va fi interpretat conținutul acestei zone. Limbajul C oferă de asemenea un mecanism de conversie a tipului care va fi prezentat mai târziu în lucrare.

Tipurile de bază ale limbajului C pot fi grupate în trei mulțimi:

- Tipuri întregi: char, short, int şi long. Aceste tipuri sunt utilizate pentru reprezentarea numerelor întregi. Diferenîa dintre acestea constă în dimensiunea zonei de memorie ocupată de o variabilă de acest tip şi implicit de mărimea domeniului de valori. Acest lucru este sintetizat în Tabela 1.

Tabela 1. Tipurile întregi

Tipul	**Dimensiunea memoriei ocupate**	**Domeniul**
char	8 biți	- 128..127
short	16 biți	- 32768..32767
int	16 biți	- 32768..32767
long (long int)	32 biți	-2147483648.. 2147483647

- Tipuri reale: float, double
  Aceste tipuri sunt folosite pentru a reprezenta numere reale. Ele se disting prin precizia de reprezentare: float – numere reale în simplă precizie –aproximativ 6 cifre semnificative - iar double pentru numere reale în dublă precizie – aproximativ 15 cifre semnificative..

- Tipul vid: void. Este un tip folosit pentru reprezentări de dimensiune nulă. Pentru tipurile întregi se pot folosi specificările signed şi unsigned. Specificarea signed *tip* este echi-valentă cu *tip* iar unsigned *tip* are semnificația din Tabela 2.

Tabela 2. Tipuri întregi unsigned

Tipul	Dimensiunea	Domeniul
unsigned char	8 biți	0 .. 255
unsigned short	16 biți	0 .. 65535
unsigned (int)	16 biți	0 .. 65535
unsigned long (int)	32 biți	0 .. 4294967295

2.2. Literali

Literalii (sau constantele literale) sunt valori constante; ele reprezintă valori care nu pot fi modificate pe parcursul execuției unui program. în limbajul C se folosesc diferite reguli sintactice pentru reprezentarea constantelor. Tabela 3 sintetizează unele din aceste reguli.

Tabela 3. Constante literale

Constante	Exemple	Explicații
întregi	123, -457, ^423, 0 O12, O3724 ox7F, OX3, A24E 39u, o245u, 57F3u	Notația zecimală Notația octală (prefix O) Notație hexazecimală (prefix ox sau OX) Constante fără semn
întregi long	12L, 42153l, o42l 423uL, 25AfuL	Reprezentare pe 32 biți Constante fără semn pe 32 biți
flotant	3.14159, -12.03, .425 -15.2E^3, 10e-12	Reprezentare fără exponent Reprezentare cu exponent
caracter	‘a’, ‘A’, ‘^’	
şir	“şir de caractere”	

Există şi caractere speciale (de control) care sunt compuse din caracterul “\” şi din alt caracter. Acestea sunt:

‘\n’ - interlinie (linie nouă); ‘\t’ - tabulare;
‘\\’ - caracterul \ (backslash); ‘\f’ - retur de linie;
‘\o’ - caracterul NULL; ‘\’’ - caracterul ‘’’
‘\”’ - caracterul ”; ‘\r’ - retur de car;
‘\u’ - tabulare verticală; ‘\b’ - backspace;
‘\ddd’ - caracter de cod octal ddd;
‘\xhh’ - caracter de cod hexazecimal hh.

2.3. Declararea unei variabile şi a unei constante

Toate variabilele unui program trebuiesc declarate înainte de a fi utilizate. Variabilele sunt obiectele de bază ale limbajului şi au un nume care este un identificator.

Un identificator în limbajul C este un şir de caractere care trebuie să îndeplinească condiţiile:

- primul caracter este literă sau '_' (underscore);
- următoarele caractere sunt caractere alfanumerice (litere sau cifre) sau '_'.
- Nu trebuie să coincidă cu vreun cuvânt cheie (vezi Tabela 4)

Tabela 4. Cuvintele cheie ale limbajelor C şi C++

asm	delete	handle	public	this
auto	do	if	register	throw
break	double	inline	return	try
case	else	int	short	typedef
catch	enum	long	signed	union
char	extern	new	sizeof	unsigned
class	float	operator	static	virtual
const	for	overload	struct	void
continue	friend	private	switch	volatile
default	goto	protected	template	while

O variabilă se declară prin una din următoarele instrucţiuni:

tip nume;
tip nume = valoare;

Mai multe variabile de acela□i tip pot fi declarate în aceea□i instruc□iune, desp□r□ite prin virgule.

O constantă se declară prin:

const *tip nume= valoare;*

Exemple:

```
int i, j, k;
float eps = 1.e-4, pi = 3.14159;
char n1='\n';
const double e = 2.71828182845905;
const float pi = 3.14159;
const char mesaj[ ] = "Eroare";
```

Variabilele se declară la începutul funcţiilor (variabile locale) sau în exteriorul tuturor funcţiilor (variabile globale). în afară de tipul variabilei şi eventual o valoare iniţială pentru aceasta, într-o declaraţie se specifică clasa de alocare (implicit sau explicit). Clasele principale de alocare sunt **automatic** şi **static**.

Variabilele din clasa automatic sunt definite în interiorul funcţiilor fiind locale funcţiei în care au fost definite, având durata de viaţă pe parcursul execuţiei funcţiei. Această clasă este implicită; explicit se poate specifica prin cuvântul cheie aut:

```
aut int x;
```

Tot în interiorul unei funcţii variabilele mai pot fi definite şi în clasa **register** care este asemănătoare cu cea **automatic** cu deosebirea că, dacă e posibil, alocarea se face în registrele unităţii

centrale şi nu în memoria RAM. Variabilele din clasele **auto** şi **register** nu îşi păstrează valoarea de la un apel la altul al funcției în care sunt definite.

Variabilele din clasa **static** diferă de cele din clasa **auto** sau **register** prin aceea că sunt memorate în locații fixe de memorie (au o adresă fixă, permanentă). Dacă se definesc în interiorul unei funcții trebuie să aibă neapărat în față cuvântul rezervat static (se mai spune că sunt în clasa **static internal**).

Variabilele definite în exteriorul tuturor funcțiilor fără vreun specificator de clasă sunt în clasa **extern** şi au alocate adrese fixe. Tabelul 5 sintetizează clasele de alocare.

Tabelul 5. Clase de alocare

CLASA	CARACTERISTICI			
	Vizibilitate	**Adresă fixă**	**Păstrează valoarea**	**Durata de viață**
Auto, Register	în interiorul funcției	NU	NU	Pe durata funcției
Static internal	în interiorul funcției	DA	DA	Pe durata aplicației
External	în toate modulele	DA	DA	Pe durata aplicației
Static external	în modulul în care-i definită	DA	DA	Pe durata aplicației
External (în funcție)	în toate modulele aplicației	DA	-	-
Static External (în funcție)	în toate modulele aplicației	DA	-	-

În capitolul al optulea sunt tratate mai în amănunt problemele legate de gestiunea memoriei.

2.4. Operatori

Operatorii se folosesc la constituirea de expresii pornind de la constante şi variabile. Evaluarea expresiilor se face înlocuind operanzii prin valorile sale şi efectuînd operațiile conform cu specificațiile acestora. Am văzut că operanzii pot fi variabile, constante sau apeluri la funcții (despre care se va vorbi mai târziu). Fiecare dintre aceştia au un tip declarat. Dacă tipul operanzilor este acelaşi într-o expresie, evaluarea se face în mod natural. Spre exemplu, dacă se declară:

```
float alpha = 20.45;
float beta = 25;
```

atunci evaluarea expresiei:

alpha + sqrt (beta),se face în ordinea indicată mai jos:

```
alpha + sqrt (25)
alpha + 5.0
20.45 + 5.0
25.45
```

Dacă operanzii au tipuri diferite , se aplică conversii implicite de tip: unul din operanzi se converteşte la tipul celuilalt iar tipul rezultatului va fi tipul comun.

Operatorii ce pot apare într-un program C sunt de patru feluri:

- operatori primari de parantetizare; aceştia delimitează subexpresii şi pot schimba asociativitatea operatorilor. De pildă expresiile

  x * y +z şi x * (y + z)

vor fi evaluate diferit pentru aceleaşi valori ale variabilelor, ca şi perechea:

x + y * z ; (x + y) * z.

- operatori binari cu notație infixată adică:
  operand operator operand.

Exemplu: x + y x % y.

- operatori unari cu notație prefixată sau postfixată:
  -x, ++x, y--, x++.
- operatorul ternar condițional
  operand 1 ? *operand 2* : *operand 3*.

2.4.1. Operatori aritmetici

Operatorii aritmetici sunt aceea care se aplică operanzilor numerici. Aceştia pot fi unari sau binari iar în privința asociativității, acolo unde nu este specificat altfel, aceasta este de la stânga la dreapta.

- Operatorii unari:
  +,- plus şi minus unari; asociativitatea este de la dreapta la stânga;
  ++ autoincrementarea;
  -- autodecrementarea;

  Negația are semnificația obişnuită: schimbă semnul valorii expresiei în fața căreia se află.
  Operatorii ++ şi -- incrementează (decrementează) cu 1 valoarea unei variabile şi pot fi :
- prefixați: variabila este incrementată (decrementată) înainte de evaluarea expresiei în care se află iar asociativitatea este de la dreapta la stânga;
- postfixați: variabila este incrementată (decrementată) după evaluarea expresiei în care se află iar asociativitatea este de la stânga la dreapta..

Exemple: Fie declarația

```
int i = 10, j, k, l, m;
```

atunci:

```
j = i ++          /* j = 10 apoi i = 10 */
k = ++i/* i = 12 apoi k = 12 */
l = i--   /* l = 12 apoi i = 11 */
m = --i /* i = 10 apoi m = 10 */
```

- Operatorii binari sunt următorii

+	adunare	*	înmulțire
-	scădere	/	împărțire
%	modulo		

La aceşti operatori se adaugă funcțiile din biblioteca matematică - prezentate în Anexa – care sunt operatori cu scriere prefixată:

- funcțiile trigonometrice uzuale: sin, cos, tan, asin, acos, atan;
- funcția de ridicare la putere pow;
- funcțiile exponențială, logaritmică etc.

2.4.2. Operatorii relaționali (de comparație)

Operatorii relaționali sunt operatori binari şi permit compararea valorilor a doi operanzi (expresii) iar rezultatul evaluării este zero (dacă rezultatul este fals) sau 1 (dacă rezultatul este adevărat). Aceştia sunt operatori binari iar sintaxa lor în C este:

<	(mai mic)	<=	(mai mic sau egal)
>	(mai mare)	>=	(mai mare sau egal)
==	(egal)	!=	(diferit)

Cum rezultatul unei comparări este un număr întreg este posibilă asignarea acestuia unei variabile întregi:

```
int a, x, y
a = (x! = y);
```

în secvența anterioară, dacă x va fi egal cu y atunci a va căpăta valoarea 0 iar în caz contrar a va căpăta valoarea 1.

2.4.3. Operatori logici

Operatorii logici sunt operatori binari care se aplică unor operanzi ce au valori adevărat (1) sau fals (0) şi ei corespund operațiilor logice uzuale:

!	- negația logică cu asociativitate de la dreapta la stânga
||	- disjuncția logică (sau)
&&	- conjuncția logică (şi)

Expresia:

- *op1&&op2* are valoarea adevărat (1) dacă cei doi operanzi sunt evaluați la adevărat, iar dacă unul din ei are valoarea fals (0) atunci expresia are valoarea zero.
- *op1||op2* are valoarea zero (fals) dacă i numai dacă *op1* şi *op2* au ambii valoareea zero (fals).
- !*op1* are valoarea fals dacă *op1* este adevărat şi are valoarea adevărat dacă *op1* este fals.

2.4.4. Operatori la nivel de bit

Limbajul C dispune de şase operatori pentru tratarea informației la nivel de bit. Aceştia se pot aplica doar operanzilor de tip întreg (char, int, short, long) şi servesc la poziționarea sau interpretarea valorilor întregi ca o secvență de valori binare. Aceşti operatori sunt:

&	- conjuncție logică la nivel de bit;
|	- disjuncție logică la nivel de bit;
^	- sau exclusiv la nivel de bit;
<<	- deplasare la stânga;
>>	- deplasare la dreapta;
~	- negare la nivel de bit (operator unar).

Semnificația primilor trei operatori binari (&, |, ^) este cea naturală (aminitm că 1 reprezintă valoarea logică "adevărat" iar 0 valoarea "fals"). Exemplele următoare sunt edificatoare.

op1 =01100100
op2 =10111010

---

op1 & op2 = 00100000
op1 | op2 = 11111110
op1 ^ op2 = 11011110

De asemenea negația (~) acționează la nivelul fiecărui bit schimbând 1 =n 0 şi 0 =n 1.

~ op1 = 10011011

Operatorii binari de decalare (<< şi >>) servesc la decalarea cu un anumit număr de biți (dat de cel de-al doilea operand) la stânga sau la dreapta în cadrul primului operand. în particular se poate obține (rapid!) înmulțirea cu 2^{op2} (op1 << op2) sau cu 2^{-op2} (op1 >> op2).

Exemple:

```
x = 13;
y = x << 2;
z = x >> 2;
```

x = 0000 0000 0000 1101(= 13)
y = 0000 0000 0011 0100(= 13.4)
z = 0000 0000 0000 0011(= 13/4)

2.4.5. Operatorul de asignare

Operatorul de asignare, exprimat prin =, are ca efect evaluarea *expresiei* din dreapta semnului = şi atribuirea acestei valori *variabilei* ce se află în stânga acestui semn.

variabila = *expresie*;

Pe lângă acest tip de asignare, limbajul C permite compunerea asignării (#) cu un operator binar (op) obținându-se operatori de asignare compusă:

e1 op = e2; unde e1 şi e2 sunt expresii.

Această asignare este echivalentă cu :

e1 = (*e1*) op (*e2*);

Se obțin asignări compuse cu:

- Operatori aritmetici: +=, -=, *=, /=, %=
- Operatori logici la nivel de bit: &=, |=, ^=
- Operatori de decalare: <<=, >>=

Faptul că atribuirea este privită ca un operator binar, permite ca aceştia să formeze expresii care au atribuite valori. Astfel

v*ariabilă = expresie;*

este o expresie ce are ca valoare, valoarea expresiei din dreapta semnului # . Acest lucru permite "atribuirea în lanț"precizând că asociativitatea operatorului de asignare este de la dreapta la stânga:

var_1 = var_2 = ... = var_n = expresie;

prin care cele n variabile capătă valoarea expresiei din partea dreaptă.

Spre exemplu

```
i = j = k = l = 1;
```

este o construcție validă în C; prin aceasta cele patru variabile i, j, k, l capătă valoarea 1.

2.4.6. Operatorul condițional

Acesta este singurul operator ternar al limbajului C. Forma generală a sa este:

op1 ? *op2* : *op3*;

unde *op1,op2,op3* sunt operanzi. Dacă *op1* are valoarea adevărat, expresia este egală cu *op2* iar dacă *op1* are valoarea fals, expresia este egală cu *op3.*

Exemplu:

```
max = (a > b) ? a : b;
```

este o expresie echivalentă cu construcția:

```
if (a > b)
    max = a;
else
    max = b;
```

Operatorul condițional se notează cu ?: şi are asociativitatea de la dreapta la stânga.

2.4.7. Operatorul virgulă

Forma generală a unei expresii ce utilizează acest operator este:

$expresie_1$,$expresie_2$, ...

Expresiile sunt evaluate de la stânga la dreapta iar valoarea întregii expresii este valoarea (şi tipul, se înțelege) expresiei ultime ce apare în acest şir.

O asemenea construcție apare frecvent în instrucțiuni for pentru realizarea mai multor acțiuni:

for(i = 0,j = 0, i < 100; i++, j++)
for(i = 0,j = n-1, i < j; i++, j--)

2.4.8. Precedența operatorilor

Asociativitatea şi precedența operatorilor indică ordinea în care se evaluează subexpresiile într-o expresie. Pentru limbajul C, operatorilor unari, operatorul condițional şi cel de atribuire li se aplică asociativitatea la dreapta pe când tuturor celorlalți li se aplică asociativitatea la stânga. Am văzut că operatorii primari de parantetizare pot schimba asociativitatea operatorilor. Precedența operatorilor limbajului C este dată în tabelul nr. 6.

Tabelul nr.6.

Operator	Funcția (semnificația)	Descrierea
:: ::	Acces explicit la o variabilă globală (unar) Acces explicit la o clasă (binar)	
→ , . [] ( ) sizeof	Selectarea unui membru Indexarea unui tablou Apel la o funcție Lungimea în biți	
-> , . ~ ! +, - * & () new,delete	Incrementare, decrementare Negare pe biți Negare logică Plus, minus unari Acces la o variabilă Adresa unei variabile Conversie de tip (cast) Operatori de gestiune a memoriei	
->* , .*	Selectarea unui membru	
*, /, %	Operatori multiplicativi	
+ , -	Operatori aritmetici	
>> , <<	Operatori de decalare	
<,<=, >, >=	Operatori relaționali	
==, !=	Egalitate, inegalitate	
&	Conjuncție pe bit	
^	Sau exclusiv pe bit	
|	Sau (disjunctiv) pe bit	
&&	Conjuncție logică	
||	Disjuncție logică	
?:	Afectare condițională	
=,*=,/=, %= +=, -=, <<=, >>= &=, ^=, |=	Operatori de asignare compusă	
,	Operatorul virgulă	

Cap 3 FUNCTII

3.1. Definirea unei funcţii
3.2. Returnarea unui apel
3.3. Funcţii cu un număr variabil de parametri
3.4. Sfârşitul execuţiei unui program
3.5. Apelul şi transmiterea parametrilor
3.6. Funcţia principală main

În limbajul C - spre deosebire de Pascal sau alte limbaje - nu există noţiunile distincte de procedură şi de funcţie. În C există doar funcţii care returnează o valoare. Este posibilă definirea de funcţii care să retuneze valoarea vidă: în acest caz tipul valorii returnate este void.

În general, utilizarea unei funcţii permite:

- descompunerea unei secvenţe lungi de procesare a informaţiei într-o mulţime de secvenţe "mici" de transformări de bază ale informaţiei;
- furnizarea către alţi programatori a unor elemente de bază de nivel înalt; detaliile acestor elemente rămân "ascunse" pentru utilizator. Aceste elemente de nivel înalt oferă o claritate în plus programului global şi permite - relativ uşor - modificări ulterioare ale acestuia.

3.1. Definirea unei funcţii

Definiţia unei funcţii are următoarea sintaxă:

```
tip nume (lista parametri)
{
    declaratii (de variabile);
    instructiuni;
}
```

În această definiţie, *tip* este unul din cuvintele rezervate care defineşte tipul funcţiei, respectiv tipul valorii returnate de un apel la această funcţie. Dacă această specificare lipseşte, tipul implicit al funcţiei este int. Funcţiile care returnează în programul apelant valoarea vidă se declară de tip void.

Numele unei funcţii este un identificator şi acesta trebuie să respecte aceleaşi reguli ca cele referitoare la numele unei variabile. Să precizăm că nu se poate defini o funcţie în interiorul altei funcţii.

Mulţimea funcţiilor unui program poate fi partiţionată în mai multe fişiere care pot fi compilate separat. Limbajul C permite de asemenea utilizarea unor funcţii predefinite (printf, getc, etc.), funcţii ce se găsesc în bibliotecile sistemului. Apelurile la aceste funcţii sunt gestionate în faza de editare a legăturilor.

Aşa cum o variabilă trebuie declarată înainte de a fi utilizată într-o expresie, o funcţie trebuie declarată înainte de a fi apelată. Declararea unei funcţii constă în a da *signatura* funcţiei. Signatura unei

funcţii defineşte atât tipul său (tipul valorii returnabile) cât şi numărul şi tipul argumentelor sale. Există două moduri de a declara o funcție:

- implicit: în definiția funcției, linia de început conține tipul său şi argumentele sale (număr şi tip);
- explicit: printr-o instrucțiune ce descrie signatura sa în cazul în care se face apel la o funcție descrisă într-un fişier separat sau în cazul în care apelul la funcția respectivă precede declarația funcției în acelaşi fişier.

Iată câteva instrucțiuni de declarare a unei funcții:

```
long int cod (const char *, int);
char* decod (long int);
void schimbare (int  , int);
int strlen (const char* s);
int funct ( );
```

În ultima linie avem un exemplu de funcție fără parametri (numărul parametrilor este zero).

3.2. Returnarea unui apel

Sintaxa instrucțiunii de returnare a valorii unei funcții este:

```
return expresie;
```

O funcție f returnează o valoare în corpul funcției din care s-a făcut apel la f, prin instrucțiunea return (plasată în corpul lui f). Valoarea expresiei *"expresie"* va fi returnată şi poate fi utilizată în programul apelant. Tipul acestei expresii trebuie să fie acelaşi cu tipul funcției. O funcție care returnează valoarea vidă (funcție de tip void) poate conține instrucțiunea "return" fără argument sau poate omite această instrucțiune.

3.3. Funcții cu un număr variabil de parametri

Există funcții care pot avea un număr nedeterminat de parametri. Un exemplu la îndemână îl oferă funcțiile sistemului: printf, scanf, etc.:

```
printf ("Un apel cu un singur argument \n");
printf("x\%f\n",x);/*Un apel cu 2 argumente */
printf ("a\%d,b\%d\n",a,b); /*3 argumente */.
```

Definirea unor astfel de funcții se face printr-o sintaxă de forma:

```
int printf (char *, ... );
```

care în C++ se numeşte "elipsă".

3.4. Sfârşitul execuției unui program

```
exit  (valoare);
```

Funcția "exit" este o funcție sistem care termină execuția unui program. Acesta este singurul exemplu de apel la o funcție care nu returnează nimic: sistemul opreşte execuția programului şi raportează valoarea

transmisă la apel. Această valoare poate fi utilă atunci când programul este executat în interiorul unei linii de comandă a interpretorului de comenzi. Prin convenție, valoarea nulă (0) semnifică faptul că programul se termină normal iar o valoare nenulă (1 sau -1) semnifică faptul că s-a produs o eroare. În absența unei instrucțiuni exit, valoarea implicită a codului returnat la sfârşitul execuției unui program este 0.

3.5. Apelul şi transmiterea parametrilor

Apelul unei funcții se face prin:

nume (*listă _ parametri_actuali*);

dacă funcția este declarată fără tip şi prin folosirea numelui funcției urmat de lista de parametri actuali într-o expresie (în care este permis tipul ei) atunci când funcția este declarată cu tip .

Ordinea evaluării parametrilor este în general de la stânga la dreapta dar acest lucru nu este garantat de orice compilator, încât trebuiesc luate măsurile corespunzătoare. De pildă, secvența:

```
int n = 10;
printf ("%d %d \n", n++, n);
```

tipăreşte (cu unele compilatoare) 10 10 şi nu 10 11.

Să precizăm că în C nu există decât un singur mod de transmitere a parametrilor şi anume *transmiterea prin valoare.* Asta înseamnă că parametrul actual (de la apel) şi parametrul formal corespunzător (declarat în antetul funcției) sunt două variabile distincte. Modificările efectuate asupra parametrilor formali nu au decât un efect local care nu este vizibil în afara funcției. Asta înseamnă că o funcție nu poate modifica parametrii actuali..

Un exemplu clasic al consecinței acestui mod de transfer al parametrilor este ilustrat mai jos:

```
void schimba1(int x, int y){
  int temp = x;
  x = y,
    y â temp;
}
main ( ){
  int a = 1, b = 2;
  schimba1(a,b);
  printf ("a = %d, b =%d", a, b);
}
```

Apelul funcției schimba1 în funcția main nu are efectul scontat: prin apelul la printf se tipăresc valorile 1 şi 2 pentru a respectiv b. Pentru înlăturarea acestui efect este posibilă simularea unui mod de *transmitere prin referință,* punând drept parametri actuali valorile adreselor variabilelor ce trebuiesc modificate. În C adresa unei variabile poate fi specificată într-o variabilă de tip "pointer" (de care vom vorbi mai târziu). Funcția schimba2 care simulează transmiterea parametrilor prin referință este:

```
void schimba2(int* x , int* y){
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

iar apelul în funcția main se face prin:

```
schimba2 (&a , &b);
```

Să facem precizarea că şi în cazul funcției schimba2 transmiterea parametrilor se face prin valoare: am simulat aici transferul prin referință ce ar fi trebuit să se facă în cazul funcției schimba1. Singura

diferență este că valoarea transmisă prin parametrii actuali &a,&b este adresa lui a respectiv b şi prin funcția schimba2 se modifică valorile referențiate. Aşadar, pentru schimbarea conținutului unor variabile printr-o funcție, parametrul corespunzător trebuie să fie de tip pointer. Argumentele de tip pointer vor fi utilizate de asemenea pentru transmiterea ca parametru a unei variabile de tip structură (vezi Cap.7) precum şi pentru limitarea numărului de valori ce trebuiesc puse în stiva de apel a execuției unui program. În capitolul 6 vom discuta mai pe larg modul de transmitere al parametrilor de tip tablou ca argumente ale unei funcții.

3.6. Funcția principală main

```
int main(int argc,char* argv [ ])
{
  instructiuni;
}
```

Funcția main este singura funcție absolut necesară într-un program C. Aceasta defineşte punctul de intrare în execuția programului. Parametrii funcției main permit recuperarea argumentelor transmise din momentul execuției programului. Valoarea întreagă returnată de funcția main este valoarea transmisă mediului la sfârşitul execuției programului (vezi funcția exit). Utilizarea instrucțiunii return precizează condiția de terminare (0 - succes, 1- eroare).

Parametrii disponibili la apelul funcției main sunt:

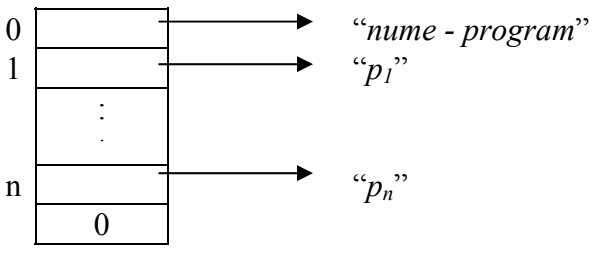
- **argc:** numărul argumentelor din linia de comandă (inclusiv numele programului);
- **argv:** un tablou de pointeri către fiecare din argumentele liniei de comandă; în particular, arg[0] conține numele programului executabil.

În cazul în care programatorul nu doreşte să utilizeze argumentele (parametrii) transmise la execuție, este permisă utilizarea funcției main fără parametri: int main( ) este o declarație validă pentru funcția main.

În general, la lansarea unui program, linia de comandă conține numele programului urmat de o listă de parametri:

nume-program $p_1, p_2, ..., p_n$

atunci argc va avea valoarea n^1 iar argv va fi un tablou ce conține pointeri astfel:



Iată un exemplu de utilizare a acestor argumente :

Programul 2

```
main (int argc, char* argv[ ] ){
 if (argc<2)
        printf ("Nu exista argumente in aceasta executie ! \n");
 else {
        printf ("Argumentele  executiei sunt: \n");
        for (i â 1; i < argc; i++)
```

```
                printf ("%s", argv [i] );
        }
}
```

Dacă acest program are numele *prg2* atunci linia de comandă:

prg2

produce scrierea mesajului:

Nu exista argumente in aceasta executie !

iar linia de comandă:

prg2 Iata niste argumente !

produce scrierea mesajului:

Iata niste argumente !

Cap 4 FUNCȚII DE INTRARE IEŞIRE

4.1 Fluxuri şi fişiere
4.2 Accesul la fişiere. Tipul FILE
4.3 Funcţii de intrare /ieşire pentru caractere
4.4 Scrierea cu format
4.5 Citirea cu format

4.1.Fluxuri şi fişiere

Sistemul de fişiere al limbajului C este conceput pentru a lucra cu o mare varietate de dispozitive, inclusiv cu terminale, unităţi de disc sau de bandă etc.. Deşi fiecare dintre acestea este foarte diferit de celelalte, sistemul de fişiere le transformă pe fiecare într-un dispozitiv logic numit *flux*. Fluxurile sunt independente de dispozitivele iniţiale , încât aceeaşi funcţie care scrie date pe un fişier de pe disc poate fi folosită pentru a scrie date pe o altă categorie de dispozitive: consola, imprimanta. Aşadar dispozitivul fizic pe care se scriu datele se numeşte fişier iar abstractizarea acestuia se numeşte *flux* (sau *stream*). Există două tipuri de fluxuri:

•*Fluxul text* este o scvenţă de caractere. Standardul ANSI C permite organizarea unui flux text în linii care se încheie cu un caracter linie nouă(“ăn”). Acesta din urmă este opţional pe ultima linie şi este determinat de modul de implementare. Datorită posibilelor conversii s-ar putea ca numărul de caractere scrise/citite să difere de numărul de caractere existente pe dispozitivul extern.

•*Fluxul binar* este o secvenţă de octeţi cu corespondenţă biunivocă faţă de octeţii existenţi pe dispozitivul extern: numărul de octeţi scrişi/citiţi este acelaşi cu numărul acestora de pe dispozitivul extern.

Un fişier în limbajul C poate fi un terminal, o imprimantă sau “o zonă” (fişier) pe disc. Un flux se poate asocia unui fişier executând o operaţie de deschidere: odată deschis, pot avea loc transferuri de informaţii între fişier şi programul curent în execuţie. Unificarea sistemului de intrare/ieşire în C se face prin “pointerii” de fişier. Un pointer de fişier este un pointer către informaţia care defineşte diverse caracteristici ale unui fişier(nume, stare, poziţie curentă etc). Pointerul de fişier identifică un anumit fişier şi este folosit de către fluxul asociat pentru a conduce operaţia de intrare/ieşire.

Pentru a putea utiliza într-un program funcţiile standard de intrare/ieşire trebuie scrisă la începutul acestui program o directivă "include" care are ca argument numele fişierului antet ce conţine definiţiile acestor funcţii: stdio.h .

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

Orice program C începe execuţia prin deschiderea unităţilor de intrare/ieşire standard:

```
stdin    : unitatea standard de intrare
stdout   : unitatea standard de ieşire
stderr   : unitatea de eroare
stdprn   : unitatea standard de ieşire prn (imprimantă)
```

Aceste unități standard sunt asociate, în lipsa altor indicații, tastaturii calculatorului (intrare) şi ecranului (ieşire şi eroare). Sistemul de fişiere ANSI C este compus din mai multe funcții legate una de alta. Cele mai frecvent folosite dintre acestea sunt date în Tabelul 5 şi vor fi prezentate în continuare.

Tabelul 5

NUME	OPERA}IE
fopen( )	Deschide un fişier
fclose( )	Închide un fişier
putc( )	Scrie un caracter într-un fişier
fputc( )	Analog lui putc()
puts( )	Scrie un şir pe ecran
fputs( )	Scrie un şir pe fişier
getc( )	Citeşte un caracter dintr-un fişier
fgetc( )	Analog lui getc( )
gets( )	Citeşte un şir de la tastatură
fgets( )	Citeşte un şir din fişier
fseek( )	Caută un anumit octet într-un fişier
printf( )	Scrie la ieşirea standard
fprintf( )	Analogul pentru fişiere al funcției printf( )
scanf( )	Citeşte din intrarea standard
fscanf( )	Analogul pentru fişiere al funcției scanf( )
feof( )	Returnează "adevărat" dacă se atinge sfărşitul fişierului
ferror( )	Returnează "adevărat" dacă survine o eroare
rewind( )	Inițializează indicatorul de poziție al fişierului la început
remove( )	Şterge un fişier
fflush( )	Goleşte conținutul unui fişier

4.1. Accesul la fişiere. Tipul FILE

Conceptul de bază pentru intrări/ieşiri standard este cel de pointer la fişier. Bibliotecile standard C oferă o serie de funcții pentru operații de intrare/ieşire orientate pe fişiere. Dispozitivele standard de intrare (tastatura) şi de ieşire (ecranul) sunt văzute tot ca fişiere cu pointerii stdin şi stdout. Aceste fişiere (stdin, stdout) pot fi orientate şi către alte dispozitive sau fişiere.

Declararea unei variabile de tip "*pointer la fişier*" se face folosind tipul FILE definit în fişierul antet stdio.h şi are forma:

```
FILE* pointer_la_fişier
```

unde *pointer_la_fişier* este un identificator (numele pointerului).

Deschiderea unui fişier se face prin funcția **fopen** care returnează ca rezultat un pointer la un descriptor de fişier. Acest dispozitiv este gestionat de sistem pentru a asigura intrările şi ieşirile efectuate în program.

```
FILE* fopen(const char* fişier,const char* mod)
```

Aici, *fişier* este un şir de caractere (identificator) ce desemnează numele extern al fişierului iar *mod* este un şir de caractere ce descrie modul de deschidere al fişierului. Modurile posibile de deschidere (deci şirurile de caractere ce constituie parametrul lui fopen) sunt date în Tabelul 6.

Tabelul 6.

mod	*Semnificația*
"r"	Citire
"w"	Creare (schimbare) pentru scriere
"a"	Adăugare la sfârşitul fişierului (existent)
"r+"	Actualizare (citire/scriere)
"w+"	Schimbare pentru actualizare
"a+"	Actualizare cu scriere la sfârşitul fişierului (existent)

Facem precizarea că, deschiderea unui fişier existent în modurile "w" sau "w+" produce pierderea conținutului său de până atunci. Dacă funcția fopen returnează o valoare nenulă, aceasta indică faptul că deschiderea s-a efectuat corect. Valoarea returnată este un pointer la un descriptor de fişier valid, valoare ce va fi utilizată în apelul la funcțiile de intrare/ieşire. Dacă deschiderea nu s-a efectuat corect valoarea returnată este NULL(pointer la informația nulă) . Acest lucru poate fi folosit pentru tratarea erorilor la deschiderea fişierului.

Închiderea unui fişier se face prin apelul la funcția fclose:

```
int fclose (FILE*  pointer_la_fișier)
```

Funcția returnează EOF în caz de eroare şi 0 în caz normal. Prin închidere se eliberează descriptorul de fişier, se actualizează fişierul pe disc şi încetează conexiunea logică între pointer şi fişier.

Apelul la fclose se poate face prin:

```
int cod;
FILE*  pointer_la_fișier

cod = fclose  (pointer_la_fișier)
```

Fişierele pot fi reasignate prin program folosind funcția freopen:

```
FILE* freopen (const char*  nume_fisier,
        const char*  mod,  FILE*  pf) ;
```

Apelul la această funcție închide fişierul cu pointerul *"pf*", deschide fişierul cu numele *"nume_fişier"* în modul de acces *"mod",* atribuind pointerul *"pf"* la acest fişier. Aceasta permite redirecționa-rea dispozitivelor periferice prin program. Programul următor ilustrează acest fapt.

Programul 3

```
# include <stdio.h>
main ( ){
  printf ("Mesaj scris pe monitor \n");
  if(freopen("iesire.dat","w+",stdout)==NULL) {
    fprintf (stderr, "Eroare la reasignare \n");
    exit (1);
  }
  printf ("Mesaj scris in fisierul iesire.dat \n");
  fclose (stdout);
  if (freopen ("CON", "wt", stdout) == NULL) {
    fprintf (stderr, "eroare la reasignarea lui       stdout la CON\n");
    exit (1);
```

```
  }
  printf ("Mesaj scris din nou pe monitor \n");
}
```

Limbajul C dispune şi de posibilitatea:

- ştergerii de fişiere prin funcţia remove:

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

- redenumirii unui fişier:

```
int rename (const char*  nume_nou,
        const char*  nume_vechi );
```

- creării unui fişier temporar:

```
FILE*tmpfile (void);
```

4.2. Funcţii de intrare/ieşire pentru caractere

```
int fgetc(FILE* flux);

char c;
FILE* pointer_la_fişier;
```

c = fgetc(*pointer_la_fişier*);

Funcţia fgetc returnează valoarea întreagă a caracterului următor din fişierul pointat cu *pointer_la_fişier* sau constanta EOF în caz de eroare sau sfârşit de fişier.

Funcţia fputc scrie un caracter la sfârşitul fişierului pointat prin *pointer_la_fişier*. În caz de eroare este returnată valoarea EOF. Există de asemenea două funcţii getchar şi putchar care permit exprimarea în C, simplificată, a scrierii apelului la fgetc şi fputc în cazul în care intrarea respectiv ieşirea este cea standard : stdin respectiv stdout.

```
int fputc(char c, FILE* flux);

int cod;
char c;
FILE* pointer_la_fişier;
```

cod = fgetc(c, *pointer_la_fişier*);

Spre exemplu:

c = getchar ( ); este echivalentă cu:
c = fgetc (stdin);

iar putchar (c); este echivalentă cu:
fput (c, stdout);.

4.3. Scrierea cu format

Funcţiile de scriere cu format permit afişarea sub forma unui şir de caractere a diverselor valori. Acestea pot fi transmise la ieşirea standard, într-un fişier sau într-un şir de caractere.

```
int printf(const char* şir_format, ...);
int fprintf(FILE* flux,const char* şir_format,...);
int sprintf(char* s,const char* şir_format,...);
```

Funcţia printf se utilizează pentru afişarea la ieşirea standard, fprintf pentru scrierea într-un fişier, iar sprintf pentru scrierea într-un şir de caractere.

Formatul *"şir-format*" este un şir de caractere ce conţine fie caractere ordinare fie directive de conversie pentru transformarea valorii unei variabile într-un şir de caractere

Caracterele ordinare sunt transmise la ieşire aşa cum apar ele pe când directivele de conversie indică formatul de ieşire pentru argumentele transmise în apelul funcţiei. O directivă se specifică prin caracterul % urmat de o secvenţă de caractere ce identifică formatul. Această secvenţă conţine, în ordine:

- eventual unul sau mai mulţi indicatori ce modifică semnificaţia conversiei. Indicatorii disponibili în C sunt:

  - : rezultatul conversiei se va cadra la dreapta;
  + : la formatul de ieşire va fi ataşat semnul + sau **-** cu excepţia cazului când primul caracter nu este semn;
  #: desemnează un format alternativ. Dacă formatul este octal se va adăuga un o înaintea numărului; dacă acesta este hexazecimal se va adăuga ox sau OX înaintea numărului. Pentru formatele flotante (e, E, f, g, G) rezultatul se va afişa cu un punct zecimal; în plus , la g sau G se vor afişa şi zerourile nesemnificative de după punctul zecimal.

- eventual un număr pentru a indica dimensiunea minimală a câmpului în care se va afişa. Dacă dimensiunea rezultată prin conversie este mai mică, caracterele se vor cadra la stânga (implicit) sau la dreapta (pentru -). Se pot utiliza şi dimensiuni variabile: se utilizează '*' în loc de numărul ce indică dimensiunea iar variabila respectivă se pune în lista de argumente.
- eventual caracterul l care precizează că numărul ce urmează a fi afişat este un "întreg lung" (long integer).
- caracterul ce indică formatul conversiei. Formatele ce se folosesc în C sunt date în Tabelul 7.

Tabelul 7. Formatele de conversie utilizate de printf.

CARACTER	TIP ARGUMENT	CONVERSIA
d sau i	int	întreg în notaţie zecimală cu semn;
o	int	întreg în notaţie octală fără semn;
x, p sau X	int	întreg în notaţie hexa fără semn; pentru x sau p se folosesc literele a, b, c, d, e, f, iar pentru X literele A, B, C, D, E, F;
u	int	întreg în notaţie zecimală fără semn
f	float sau double	real în notaţie zecimală sub forma [-]mmm.dddddd sau numărul de d este dat de argumente de precizie (implicit acesta este 6);

e sau E	float sau double	real în notație zecimală sub forma: [-]m.dddddde ± xx sau [-]m.ddddddE ± xx. Numărul de d este dat de argumente de precizie (implicit este 6);
g sau G	float sau double	afişare în acelaşi stil cu e dacă exponentul este -4 sau superior preciziei; în stil f în caz contrar;
c	int	afişarea unui singur caracter după conversie în unsigned char;
s	char*	afişarea unui şir de caractere până la întâlnirea caracterului '@0' (sfârşit de şir) sau până când numărul de caractere este egal cu numărul de precizie;
p	void*	afişează argumentul în notația pointer;
n	int*	nu există nici conversie nici afişare în acest caz: este vorba de referințe la un pointer la un întreg în care se va stoca numărul de caractere afişate până în prezent;
%		Permite afişarea caracterului ‘ % ‘

Exemple

Următoarele programe ilustrează folosirea formatelor de afişare la apelul funcției printf rezultatele sunt ilustrate la sfârşitul fiecărui program.

Programul 4

```
# include <stdio.h>
int main ( )
{
  int                 i = 123;
  long int            l = 123456789L;
  unsigned int        ui = 45678U;
  unsigned long int   ul = 987654321UL;
  double              d = 123.45;
  printf(" Forma zecimala:\n");
  printf("   i = %d\n",      i);
     printf("   l = %ld\n",   l);
printf("  ui = %u\n",   ui);
printf("  ul = %lu\n\n", ul);

printf(" Forma octala:\n");
printf("  ui = %o\n",   ui);
printf("  ul = %d\n", i);
printf("   i = %lo\n\n",ul);
```

In urma execuției se va afişa:

```
Forma zecimala:
    i = 123
    l = 123456789
   ui = 45678
   ul = 987654321
```

Forma octala:
  ui = 131156
  ul = 7267464261

```
printf("Scrierea hexazecimala:\n");
printf("   ui = %X\n",      ui);
printf("   ul = %lX\n\n",   ul);

printf("Afisarea semnului:\n");
printf("|% d| |%+d| |%d|\n",-123,-123,-123);
printf("|% d| |%+d| |%d|\n", 123, 123, 123);

printf("Afisare in octal si hexazecimal:\n");
printf("|%x| |%#x|\n", 123, 123);
printf("|%X| |%#X|\n", 123, 123);
printf("|%o| |%#o|\n", 123, 123);
printf("\n");
```

În urma execuției se va afișa:

```
Scrierea hexazecimala:
  ui = B26E
  ul = 3ADE68B1

Afisarea semnului:
  |-123| |-123| |-123|
  | 123| |+123| |123|

Afisare in octal si hexazecimal:
  |76| |0x7b|
  |7B| |0X7B|
  |173| |0173|
```

```
/*Specificarea lungimii campului si a
**numarului de cifre*/
printf("|%.4d|\n",  123);
printf("|%+.4d|\n",          123);
printf("|%.4X|\n", 123);
printf("\n");

printf("|%5d| |%-5d|\n",    123, 123);
printf("|%+5d| |%+-5d|\n",123, 123);
printf("|%#5d| |%#-5d|\n",123, 123);
printf("\n");

printf("|%+6.4d|\n",        123);
printf("|%+6.4o|\n",        123);
printf("|%+6.4X|\n",        123);
printf("\n");
```

Iată ce se va afişa:

```
|0123|
|+0123|
|0X007B|

|  123| |123  |
| +123| |+123 |
| 0X7B| |0X7B |

| +0123|
|  0173|
|0X007B|
```

```
/* Afisarea numerelor in virgula flotanta*/
printf("|%f|\n"          3.14157);
printf("|%.3f|\n"       3.14157);
printf("|%.0f|\n"       3.14157);
printf("|%#.0f|\n"      3.14157);
printf("|%E|\n"                 3.14157e123);
printf("|%E|\n"                 3.14157e123);
printf("|%.3E|\n"       3.14157e123);
printf("|%.0E|\n"       3.14157e123);
printf("|%#.0E|\n"      3.14157e123);

printf("|%f|\t|%G|\n", 3.1, 3.1);
printf("|%E|\t|%G|\n", 3.1e10, 3.110);

printf("|%G|\t|%G|\n", -0.0001, 0.00001);
printf("|%f|\t|%G|\n", 3.1e5, 3.1e7);
printf("|%.11G|\t|%.11G|\n", 3.1e6, 3.1e11);
return0;
}
```

Iată ce se va afişa de această dată:

```
|3.141570|
|3.142|
|3|
|3.|
|3.141570E+123|
|1.142E+123|
|3E+123|
|3.E+123|
|3.100000|              |3.1|
|3.100000E+10|          |3.1E+10|
|-1.0001|               |1E-05|
|310000|                |3.1E+07|
|3100000|               |3.1 E+11|
```

4.5. Citirea cu format

Funcțiile de citire cu format permit interpretarea unui şir de caractere ca o secvență de valori în concordanță cu un format dat. Rezultatul interpretării este încărcat la adresele ce se dau ca argumente la apelul funcțiilor de citire. Aceste funcții, corespunzătoare citirii de la intrarea standard, de la un fişier respectiv dintr-un şir de caractere sunt:

int scanf (const char* *şir_format,...*);
int fscanf (FILE* *flux*, const char* *şir_format,...*);
int sscanf (char* s, const char* *şir_format*, …);

Formatul (*şir_format*) este un şir de caractere care poate conține:
- caractere de spațiere sau tabulare care vor fi ignorate;
- caractere ordinare (în afară de %) care trebuie să corespundă următorului caracter citit;
- directive de specificare a formatului, care încep cu '%'.

Directivele pentru specificarea unui format sunt utilizate pentru precizarea conversiei următorului câmp citit. În general, rezultatul citirii este memorat în variabila corespunzătoare din lista argumentelor funcției. Se poate face o omisiune de afectare, punând caracterul '*' în directivă (de ex. %*s): în acest caz câmpul este citit dar nu-i memorat.

Câmpul ce urmează a fi convertit (ce este citit) se întinde până la primul spațiu sau numărul de caractere din acest câmp este specificat în format.

Caracterele pentru specificarea unui format de citire sunt date în Tabelul 8.

Tabelul 8. Formatele de conversie pentru scanf

CARACTER	FORMAT DE INTRARE	TIPUL ARGUMENTULUI
d	întreg în notație zecimală;	int*
i	întreg în notație octală(primul caracter 'o') sau hexa (primul caracter 'Ox' sau 'OX');	int*
o	întreg în notație octală;	int*
u	întreg fără semn în notație zecimală;	unsigned int*
x	întreg în notație hexa;	int*
c	caracter. Implicit se citeşte un singur caracter. Dacă este indicat un număr q (%qC) atunci sunt citite cel mult q caractere, citindu-se şi spațiile care, de obicei sunt ignorate. Pentru citirea următorului caracter diferit de spațiu se foloseşte %ls.	char*
s	şir de caractere: se citeşte până la întâlnirea unui spațiu sau până la atingerea numărului de caractere indicat;	char*
e, f, g	real în notație virgulă flotantă;	float* double*
p	valoarea unei adrese;	void*
n	scrie în argument numărul de caractere citite până acum. Nu se face nici o operație de citire;	int*

[...]	citeşte caracterele din intrare atât timp cât acestea aparţin mulţi-mii indicate în [ ]. Adaugă la argument caracterul '@0' (sfârşit de şir);	char*
[^...]	citeşte caracterele din intrare atât cât acestea nu sunt în mulţimea indicată în [ ] după care adaugă '@0';	char*
%	permite indicarea caracterului literal '%'.	-

Programul următor ilustrează utilizarea formatelor de intrare.

Programul 5

```
#include <stdio.h>
int main(){
    int         i1, i2, i3;
    charc1, c2, c3;
    chars1[10],s2[10],s3[10];
    char        s4[5], s5[5], s6[5];
    printf("Introduceti trei numere intregi: ");
    scanf("%d%d%d", &i1, &i2, &i3);
    printf("Ati introdus:%d\t%d\t%d\n",i1,i2,i3);

    printf("Introduceti trei numere intregi: ");
    scanf("%4d%3d%4d", &i1, &i2, &i3);
    printf("Ati introdus:%d\t%d\t%d\n",i1,i2,i3);

    printf("Introduceti un text: ");
    scanf("%s%s%s, s1, s2, s3");
    printf("s1=|%s|\ts2=|%s|\ts3=|%s|\n",s1,s2,s3);

    printf("Introduceti un text: ");
    scanf("%4s %4s %4s", s1, s2, s3);
    printf("s1=|%s|\ts2=|%s|\ts3=|%s|\n",s1,s2,s3);
    scanf("%c%c%c", &c1,&c2,&c3);
    printf("c1= |%c|\tc2=|%c|\tc3=|%c|\n"c1,c2,c3);
    printf("Introduceti un text: ");
    scanf("%4c%4c%4c", s4,s5,s6);
    s4[4] =s5[4] =s6[4] ="\0";
    printf("s4=|%4s|\ts5=|%4s|\ts6=|%4s|\n",s4,s5,s6)return 0;
}
```

Rezultatul unei execuţii a acestui program este:

```
Introduceti trei numere intregi: 5454 -9988 0
A-ti introdus numerele: 5454 -9988 0
Introduceti trei numere intregi: 5454998 390
A-ti introdus numerele: 5454 998 390
Introduceti un text: Iata un exemplu!
s1=|Iata|       s2=|un| s3=|exemplu!|
```

```
Introduceti un text: Iata un exemplu!
s1=|Iata|          s2=|un| s3=|exem|
c1=|p|  c2=|l|   c3=|u|
Introduceti un text: Iata un exemplu!:
s4=|!
Ia| s5=|ta u|      s6=|n ex|
```

Cap 5 INSTRUCȚIUNI DE CONTROL

5.1. Instrucțiunile if
5.2. Instrucțiunea switch
5.3. Instrucțiunea while
5.4. Instrucțiunea do...while
5.5. Instrucțiunea for
5.6. Instrucțiunea break
5.7. Instrucțiunea continue
5.8. Instrucțiunea go to
5.9. Exerciții
5.10. Soluții la exerciții

Limbajul C oferă următoarele tipuri de instrucțiuni:

- declarații;
- expresii;
- instrucțiuni de control;
- blocuri de instrucțiuni;
- definiții de funcții.

În capitolul 2 am văzut cum se construiesc expresiile în C. O expresie urmată de caracterul ';' este o instrucțiune în C. În general o instrucțiune se termină prin ';'. Următoarele construcții sunt instrucțiuni C de tip expresii:

```
i = 1;
j ++;
++ k;
cod = printf ("%d\n", alpha);
```

Un bloc este un șir de instrucțiuni încadrat în acolade: { și } (care joacă rolul de begin ... end din alte limbaje). Un bloc este echivalent cu o instrucțiune. Iată un bloc în C:

```
{
  i = 1;
  j = 2;
  s [i] = t [j];
  i ++;
  j --;
  s [i] = t [j];
}
```

Instrucțiunile de declarare a variabilelor (declarațiile) sunt de forma(vezi și paragraful 2.3) :

```
tip  identificator;
tip  identificator  = valoare;
```

Vom trece în revistă în continuare instrucțiunile de control ale limbajului C.

5.1. Instrucțiunea if

```
if (expresie)
   instrucțiune1
else
   instrucțiune2
```

```
if ( expresie )
   instrucțiune1
```

Ca și în alte limbaje de programare această instrucțiune permite alegerea între două alternative. Tipul expresiei de după cuvântul rezervat if poate fi întreg (char, short, int sau long), real (float sau double) sau tip pointer. Expresia se pune obligatoriu între paranteze iar partea "else *instrucțiune2*;" este facultativă. Semantica acestei instrucțiuni este uzuală: dacă valoarea expresiei "*expresie*" este nenulă (ceea ce înseamnă "true") atunci se execută *instrucțiune1* iar dacă această valoare este nulă ("false") se execută *instrucțiune2* (dacă aceasta există).

Exemple:

```
if (x>y)
        x = x-y;
else
        y = y-x;
```

```
if ((c = getchar ( ) ) ! = EOF)
        putchar (c);
```

```
if (x>y)
        printf("%d este cel mai mare\n", x);
else
        printf("%d este cel mai mare\n", y);
```

```
if (x>y)
        printf("primul numar este cel mai mare\n");
else if (x==y)
        printf("numerele sunt egale\n");
else
        printf("al doilea numar este mai mare\n");
```

```
if ( a < b ) {
        a += 2;
        b += 1;
};
else
        a -=b;
```

Atenție! În acest ultim exemplu este o eroare de sintaxă: alternativa else nu poate fi asociată cu if pentru că înaintea sa (după }) apare caracterul ';' care reprezintă instrucțiunea vidă! Apare și problema ambiguității (așa zisa problemă "*dangling else*"):

```
if ( a == 1 )
if ( b == 2 )
printf (" b este egal cu doi\n");
else
printf (" b nu este egal cu doi\n");
```

Cărui if îi este atașat else? Limbajul C rezolvă problema aceasta prin atașarea lui else celui mai apropiat if, deci exemplul este interpretat astfel:

```
if ( a == 1 )
        if ( b == 2 )
                printf (" b este egal cu doi\n");
```

```
else
        printf (“ b nu este egal cu doi\n”);
```

5.2. Instrucțiunea switch

```
switch(expresie){
        case const1:
                instrucțiune1
        case const2:
                instrucțiune2
        …
        case constn:
                instrucțiunen
        default:
                instrucțiune
}
```

Instrucțiunea switch este o structură de control cu alegeri multiple ce înlocuiește o succesiune de alternative de tip if. Aceasta permite execuția unei anumite ramuri în funcție de valoarea expresiei "*expresie*" din paranteză, care poate fi de tip întreg, caracter sau enumerare. Dacă expresia are valoarea $const_1$ atunci se execută $instrucțiune_1$ și toate cele ce urmează după aceasta până la întâlnirea unei instrucțiuni "break", dacă valoarea sa este $const_2$ atunci se execută $instrucțiune_2$ și următoarele până la "break" etc. Cazul default este facultativ; dacă expresia nu are nici una din valorile indicate prin $cons_1$, $cons_2$, ..., se execută instrucțiunile desemnate de ramura "default", dacă aceasta există.

Exemplu:

```
switch (getchar ( ) ) {
case 'a':
        printf ("s-a citit litera a\n");
        break;
case 'b':
        printf ("s-a citit litera b\n");
        break;
case 'c':
        printf ("s-a citit litera c\n");
        break;
default:
        printf ("s-a citit altceva decat a, b sau c\n");
      break;
}
```

În acest exemplu, pe fiecare ramură s-a folosit instrucțiunea break. Se înțelege că, în cazul în care există pe una din ramuri alte instrucțiuni de control, break poate lipsi (vezi exemplul de la insrtucțiunea for).

5.3. Instrucțiunea while

```
while (expresie)
      instrucțiune
```

Expresia "*expresie*", ce trebuie pusă între paranteze, este evaluată și dacă aceasta nu este nulă, se execută "*instrucțiune*" (care poate fi, binențeles, un bloc). După execuția acestei instrucțiuni se face din nou evaluarea expresiei indicate în while și se repetă execuția până când evaluarea expresiei returnează o valoare nulă.

Exemplu:

```
int c, n = 0;
while ((c = getchar( )) != EOF){
        putchar (c);
        n++
}
```

```
int x, y;
while (x!=y) {
        if (x>y)
                x = x-y;
        else
                y = y-x;
}
```

Programul următor ilustrează utilizarea instrucțiunii while și a instrucțiunii switch. Acest program analizează un text și numără vocalele, spațiile goale și restul caracterelor din acesta.

Programul 6

```
# include <stdio.h>
int main (void){
  int nvocale, naltele, nspatii;
  char c;
  nvocale = naltele = nspatii = 0;
  printf ("Introduceti un text\n -->");
  while ((c = getchar( )) != '\n') {
        switch (c) {
                case 'a': case 'A':
                case 'e': case 'E':
                case 'i': case 'I':
                case 'o': case 'O':
                case 'u': case 'U':
                        nvocale++; break;
                case ' ': case '\t':
                        nspatii++; break;
                default: naltele++;
        }
  }
  printf("In text sunt %d vocale, %d spatii si %d alte
 caractere.\n",nvocale,nspatii,naltele);
```

```
  return 0;
}
```

Rezultatul unei execuții:
Introduceti un text:
--> Iata un text pentru test !
In text sunt 8 vocale,5 spatii si 13 alte caractere.

5.4. Instructiunea do ... while

```
do
   instrucțiune
while(expresie);
```

Spre deosebire de instrucțiunea while, "corpul" instrucțiunii do...while (adică "*instrucțiune*") se execută la intrarea în această buclă după care se testează condiția "*expresie*": dacă aceasta are valoare nenulă se execută din nou "*instrucțiune*" etc. Execuția se termină atunci când "*expresie*" are valoarea nulă. Așadar în cazul do...while, "*instrucțiune*" se execută măcar o dată pe când în cazul *while*, dacă "*expresie*" este evaluată prima dată la zero, "*instrucțiune*" nu se execută.

Exemplu:

```
int n = 0, c;
do {
        c = getchar ( );
        n++;
} while (c!= '\n');
printf ("Linia contine %d caractere \n", n-1);
```

```
int n = 1; c;
c = getchar ( )
while (c!= '\n') {
        c = getchar ( );
        n++
};
printf ("Linia contine %d caractere\n", n-1);
```

Se constată ușor că cele două secvențe de program sunt echivalente.

5.5. Instrucțiunea for

```
for (expr1; expr2; expr3)
        instrucțiune
```

Semantica acestei instrucțiuni poate fi descrisă în două moduri:

1. • se execută (o singură dată) *expr1* și se intră în buclă (execuția de n ≥0 a instrucțiunii "*instrucțiune*");

• expresia *expr2* se evaluează și se testează la fiecare început de buclă: dacă este adevărată atunci se execută "*instrucțiune*" altfel bucla se termină;

• instrucțiunea *expr3* se execută la fiecare sfârșit de buclă;

2. Instrucțiunea este echivalentă cu secvența:

```
expr1;
while (expr2) {
        instrucțiune
        expr3;
}
```

Instrucțiunea poate fi folosită și prin suprimarea uneia dintre *expr1, expr2* sau *expr3* sau prin suprimarea corpului ("*instrucțiune*"):

- Suprimarea inițializării;

```
for( ; expr2 ; expr3)
        instrucțiune
```

Aceasta este echivalentă cu secvența dată mai sus (2.) din care lipsește prima instrucțiune (*expr1*;)

Exemplu:

```
i = 0;
for(;(c = getchar( )) != '\0'; i++)
        putchar (c);
printf("numarul de caractere:%d\n", i);
```

Să observăm că inițializarea lui i s-a făcut în afara instrucțiunii for !

- Suprimarea expresiei de control:

```
for( expr ; ; expr3)
        instrucțiune
```

Aceasta este o buclă infinită. Singura modalitate de a ieși din aceasta este utilizarea instrucțiunii break.

Exemplu:

```
for (i = 0; ; c = getchar ( ), i++)
        if (c == '\n')
                break;
```

- Suprimarea expresiei finale:

```
for( expr1 ; expr2 ; )
        instrucțiune
```

Exemplu:

```
for (i = 0; (c = getchar( )) != '\0';){
        putchar (c)
        i++
}
printf ("numarul de caractere: %d\n", i);
```

- Suprimarea inițializării și a expresiei finale:

```
for( ; expr2 ; )
        instrucțiune
```

Exemplu:

```
i = 0
for (; (c = getchar( )) != '\0'; {
putchar (c)
        i++
}
printf ("numarul de caractere: %d\n", i);
```

- Suprimarea corpului buclei:

```
for( expr1 ; expr2 ; expr3 ) ;
```

Exemplu:

```
for (i = 0; getchar( )!= '\n'; i++);
```

5.6. Instrucțiunea break

Instrucțiunea break permite ieșirea dintr-o buclă (for, while, do) sau dintr-o instrucțiune cu alegeri multiple (switch). Iată câteva exemple de utilizare a acestei instrucțiuni (vezi și programul de la sfârșitul paragrafului 5.3):

1.

```
while ( ( c = getchar( )) != EOF) {
        sum++;
        if (sum>=50) {
                printf ("50\n");
                break;
        }
}
printf ("Iesire din bucla while\n");
```

În acest prim exemplu, instrucțiunea break permite ieșirea din bucla while după citirea a 50 de caractere și nu la întâlnirea caracterului EOF.

2.

```
while ( ( c = getchar( ))!= EOF) {
        sum++;
        for (j = sum; sum <= 25; j++) {
                j* = 1.5;
                if ( j >= 75.5) break;
        }
        printf ("Total: %d", sum);
}
```

În acest caz instrucțiunea break permite ieșirea din bucla for.

5.7. Instrucțiunea continue

Instrucțiunea continue ignoră toate instrucțiunile ce urmează după ea, până la sfârșitul buclei în care se află; ea provoacă trecerea la următoarea iterație. Iată un exemplu:

```
int     numar, suma = 0;
for (numar = 1; numar <= 100; numar++) {
        if (numar % 5 == 0)
                continue;
        suma++ = numar;
}
print ("Suma este:%d\n", suma);
```

Se observă că prin continue se evită adăugarea la suma a numerelor multiple de 5.

5.8. Instrucțiunea go to

Instrucțiunea go to provoacă saltul necondiționat la o etichetă, în cuprinsul aceleași funcții.

go to *eticheta* ;

Etichetele sunt nume ce se pun în fața instrucțiunilor țintă urmate de ":". Este recomandat să se evite folosirea instrucțiunii go to (așa cer principiile programării structurate) dar uneori este utilă. În exemplul următor se ilustrează folosirea instrucțiunii go to și modul cum ea poate fi evitată:

```
for (i = 0; i < n; i++)
        for (j=0; j < m; j++)
                if (a [i] == b [j] )
                        go to gasit;
printf("Tablourile nu au elemente comune\ n");
        .
        .
        .
gasit: printf (" a[%d]=b[%d]=%d\n",i,j,a [i]);
```

```
int gasit = 0;
for (i = 0; i < n && ! gasit; i++)
        for (j = 0; j < m &&! gasit; j++)
                gasit = (a [i] == b [j]);
if (gasit)
        printf (" a[%d]= b[%d]= %d\n",i,j,a[i]);
else
printf ("Tablourile nu au elemente comune \n");
```

5.9. Exerciții

jgjj

1. Calculul factorialului.

Scrieți un program care să realizeze următoarele:
- citirea unui număr natural (dat de utilizator);
- calculul factorialului acestui număr (iterativ);
- afișarea rezultaului.

2. Conversie

Scrieți un program care citește un număr hexazecimal (ca o succesiune de caractere), verifică validitatea sa și efectuează conversia în zecimal. Se va utiliza o iterație pentru citirea caracter cu caracter a numărului, iterația se oprește dacă se citește un caracter ce nu este cifră hexazecimală și, pe măsură ce se citesc cifrele hexazecimale se face conversia în zecimal. Scrieți două variante ale programului:

a) programul conține doar funcția principală main ( )

b) funcția principală main face apel la o funcție de conversie a unui caracter reprezentând o cifră hexazecimală în valoarea sa zecimală:

```
int conversie (char x);
```

3. Numărarea biților 1 într-un număr

Scrieți un program care citește 10 numere întregi și pentru fiecare din acestea calculează numărul biților ce au valoarea 1. În bucla de citire, se va face apel la o funcție ce numără biții 1.

4. Prezentare de date

Scrieți un program care citește n valori întregi și le afișează câte l pe fiecare linie, adăugând la sfârșitul liniei suma lor. Ultima linie va conține suma sumelor, aliniată la sfârșit de linie. Numerele n și l se definesc ca valori constante.

5.10. Soluții la exerciții

1. Calculul factorialului

Iată o soluția incompletă a problemei. Programul va trebui completat cu un test pentru a controla valoarea citită și cu un test pentru a verifica dacă rezultatul calculului nu depășește capacitatea de reprezentare a unui număr întreg.

```
/* Calculul lui n! */
# include <stdio.h>
int  main (void)
{
        int n, i, fact;
        printf ("Introduceti un numar natural ->");
        scanf ("%d", &n);
        fact = 1;
        for (i = 2; i < = n; i++)
                fact * = i;
```

```
        printf (" %d ! = %d\", n, fact);
        return 0;
}
```

2. Conversie

Vom construi o buclă while care se execută atâta timp cât ultimul caracter citit este o cifră hexazecimală: 0-9 sau a-f sau A-F. Dacă este o astfel de cifră, se convertește în valoare întreagă, păstrând valoarea cumulată a numărului zecimal obținut. Prima soluție face conversie în funcția main ( ) iar a doua soluție folosește o funcție de conversie.

a)

```
/*Citirea unui numar hexa si conversia sa in zecimal*/
# include <stdio.h>
int main (void){
        int numar = 0;
        char c;
        printf (" Introduceti un numar hexa ->");
        c = getchar ( ) ;
        while (( c>= '0') && (c<='9')||
                (c>= 'a') && (c<='f')||
                (c>= 'A') && (c<='F') {
                if ((c>= '0') && (c<='9'))
                        numar = numar*16 + (c - '0');
                else if ((c>= 'a') && (c<='f'))
                        numar = numar*16 + (c - 'a' + 10);
                else
                        numar = numar*16 + (c - 'A' + 10);
                c = getchar ( );
        }
        printf("Conversia zecimala :%d\n", numar);
        return 0;
}
```

b)

```
/*Citirea unui numar hexa si conversia sa in zecimal*/
# include <stdio.h>
int conversie (char x){
        return (x>= '0') && ( x<= '9') ? x - '0':
                ((x>='a')&&(x<='f')?x-'a':x-'A')+10;
}
int main (void){
        int numar = 0;
        char c;
        printf (" Introduceti un numar hexa ->");
        c = getchar ( ) ;
        while (( c>= '0') && (c<='9')||
```

```
                (c>= 'a') && (c<='f ')||
                (c>= 'A') && (c<='F') {
                numar = numar*16 + conversie (c);
                c = getchar ( ) ;
        }
        printf ("Conversia zecimala: %d\n", numar);
          return 0;
}
```

3. Numărarea biților 1 într-un număr

Determinarea biților 1 se face prin decalaj succesiv și testarea bitului.

```
/* Numararea bitilor 1 din reprezentarea binara a unui numar intreg */
# include <stdio.h>
int numarbit(int);
int main (void)
{
  int i;
  int val;
  for (i = 0; i < 10; i++) {
        printf ("Introduceti o valoare intreaga: ");
        scanf ("%d", &val);
        printf ("Numarul bitilor 1 este: %d\n",
                numarbit(val));
  }
  return 0;
}
int numarbit(int x){
  int n = 0;
  while (x!=0) {
        if (x & 01)
                n++;
        x >>=1;
  }
  return n;
}
```

În funcția numarbit, transmiterea parametrului se face prin valoare; se poate așadar utiliza x ca variabilă de lucru. Prin instrucțiunea x >>=1 se decalează biții lui x cu o poziție la dreapta; la stânga se adaugă un zero.

4. Prezentare de date

```
/* Se citesc n intregi si se afiseaza cate l pe **linie impreuna cu suma acestora la sfarsit de
**linie.Ultima linie contine suma sumelor */
# include <stdio.h>
const n = 10;
const l = 3;
int main (void){
```

```
    int val,   /*ultima valoare intreaga citita */
               sumap=0,      /*suma din linia curenta*/
               nl=0,/*numarul intregilor pe linie */
               suma=0,       /* suma sumelor par]iale */
               nt;           /*numarul total de intregi */
  printf ("Introduceti%d numere intregi", n);
  for (nt = 0; nt < n, nt++) {
               scanf ("%d", &val);
               ++nl;
               suma p+= val;
               printf ("%6d", val);
               if (nl == l ) {
                        printf (" %6d\n", sumap);
                        nl = 0;
                        suma += sumap;
                        sumap = 0;
               }
  }
  if (nl! = 0)  {
            /*ultima linie este incompleta: se **completeaza cu spatii*/
               int i;
               for (i = nl; i < l; i++)
                        printf ("      ");
               suma += sumap;
               printf ("%6d\n", sumap);
      }
      /* alinierea sumei totale pe ultima linie */
      int i;
      for (i = 0; i < l; i++)
               printf ("      ");
      printf ("%d\n", suma);
      return 0 ;
}
```

Cap 6 TABLOURI ŞI POINTERI

6.1. Pointeri
6.2. Tablouri cu o dimensiune
6.3. Relația între pointeri și tablouri
6.4. Operații aritmetice cu pointeri
6.5. Șiruri de caractere
6.6. Tablouri cu mai multe dimensiuni
6.7. Tablouri de pointeri
6.8. Pointeri și alocarea memoriei
6.9. Operatorul sizeof
6.10. Pointeri către funcții
6.11. Exerciții
6.12. Soluții

6.1. Pointeri

tip ***identificator*;

Un pointer este o variabil\ ce con]ine adresa unei alte variabile. Pointerii au tip : aceasta `nseamn\ c\ un pointer nu poate fi folosit dec=t pentru a con]ine adrese de variabile de un singur tip. De exemplu, prin declara]iile:

```
int *pi;
char *pc;
```

se specifică faptul că pi este un pointer la o variabilă de tip întreg iar pc un pointer la o variabilă de tip "char".

Operatorul unar "&" numit operator de *referențiere* sau *adresare*, aplicat unei variabile, permite obținerea adresei acesteia:

```
int i;
pi = &i;
```

Operatorul unar "*", *dereferențiere, indirectare*, permite derefențierea unui pointer, furnizând valoarea variabilei pointate de acesta:

```
int x;
x = *pi ⇔ x = i;
```

Utilizarea operatorului "*" este destul de delicată, chiar periculoasă am spune: acesta permite citirea sau scrierea în orice zonă de memorie. Utilizarea unui pointer care este inițializat greșit este o eroare des întâlnită în programare și din păcate consecințele sunt imprevizibile.

O aplica]ie important\ privind utilizarea pointerilor este transferul prin referin]\ al parametrilor unei func]ii. A[a cum s-a precizat la capitolul "Func]ii", `n limbajul C transferul implicit al parametrilor este prin valoare. Dac\ se dore[te modificarea unui parametru formal, trebuie transmis\ func]iei adresa variabilei iar `n corpul func]iei s\ se foloseasc\ operatorul de dereferen]iere.

Exemplul clasic pentru acest procedeu este acela al funcției de interschimbare a două variabile. Forma "clasică" a funcției, incorectă (semantic) în limbaj C, este:

```
void schimbare (int a, int b)        {
        int temp = a; a = b; b = temp;
}
```

Un apel al acestei funcții (prin schimbare (x, y);) nu are nici un efect asupra variabilelor x,y (actuale) pentru că funcția lucrează cu variabilele ei locale a și b. Varianta corectă este:

```
void schimbare_corecta (int *p, int *q){
        int temp;
        temp = *p; *p = *q; *q = temp;
}
```

Apelul corespunzător este:

```
int x = 1, y = 2;
schimbare_corecta (&x, &y);
```

6.2. Tablouri cu o dimensiune

tip nume[*dimensiune*];

Un tablou este o colec]ie de elemente(date) de acela[i tip, fiecare din acestea putând fi accesibilă. Declarația

int vector[10];

definește un tablou cu numele "vector" care are 10 elemente de același tip, întreg. Așadar, o declarație de tablou cu o dimensiune conține tipul tabloului (tip), numele tabloului (nume) - care este un identificator - precum și dimensiunea sa, dimensiune. Dimensiunea este o constantă întreagă și specifică numărul elementelor tabloului.

6.2.1. Referențierea unui element al tabloului

Elementele unui tablou pot fi accesate prin numele tabloului urmat de o expresie între paranteze pătrate - numită expresie indice - care trebuie să aibă valoarea între 0 și dimensiune - 1. Primul element al tabloului este cel de indice 0 iar ultimul este cel de indice *dimensiune* -1. În declarația de mai sus, int vector[10]; cele 10 elemente ale tabloului vector sunt:

vector[0], vector[1],…, vector[9]

La declararea unui tablou se alocă o zonă de memorie contiguă care să stocheze toate elementele tabloului. Adresa primului element (cel de indice 0) este și adresa tabloului iar celelalte elemente se raportează la acesta.

6.2.2. Inițializarea unui tablou

Inițializarea unui tablou se face prin semnul "=" după declarația acestuia urmat de o listă de valori inițiale, separate prin virgule, care se închid între acolade.

```
tip nume[dim] = {
    valoare_1,
    valoare_2,
    ...
};
```

Exemple:

1. int a[4] = {1, 2, 3, 4};
   char s[4] = {'a', 'b', 'c', '\o'};

2.
```
# define marime  5;
char tab[marime];
tab[0] = 'a';
tab[1] = tab[2] = 'b';
tab[3] = tab[4] = 'd';
```

Tabloul va avea structura:

'a'	'b'	'b'	'd'	'd'

```
tab[marime - 1] = 'e';
tab[marime/2] = 'c';
```

Dupa aceste transformari structura sa va fi:

'a'	'b'	'c'	'd'	'e'

Dacă lista de valori inițiale (dată după declarație) cuprinde mai puține elemente decât dimensiunea declarată a tabloului, atunci celelalte valori se inițializează cu zero. Spre exemplu

int a[10] = {5, 2, 3, 1};

are ca efect crearea tabloului:

5	2	3	1	0	0	0	0	0	0

În cazul unui tablou care se inițializează, poate fi omisă dimensiunea; aceasta se determină de către compilator:

float x[ ] = {2, 0, 5.2, 3, 4, 0};

2	0	5.2	3	4	0

În cazul tablourilor de caractere, inițializarea se poate face și cu șiruri de caractere incluse între ghilimele; declarația:

char s[6] = "abcdef" ;

este echivalentă cu:

char s[6] = {'a', 'b', 'c', 'd', 'e', 'f'};

iar declarația: char s[ ] = "abcdef" ;

este echivalentă cu:

char s[ ] = {'a', 'b', 'c', 'd', 'e', 'f', '\0'};

De notat că în cazul sirurilor de caractere ce se memorează într-un tablou, ultimul element al tabloului este caracterul '\0'; acest caracter notează sfârșitul șirului(vezi paragraful 6.5).

Ini]ializarea unui tablou se poate face [i prin citirea succesiv\ a tuturor elementelor sale:

```
int i;
```

```
float x [20];
for (i=0; i < 20; i++) {
    scanf (" %f", &x[i]);
}
```

Este de remarcat faptul că nu se face nici o verificare a limitelor unui tablou atunci când se accesează un element al său. Spre exemplu dacă se declară

```
int a[3], b[3], c[3];
```

și alocarea se face astfel:

a[0]	a[1]	a[2]	b[0]	b[1]	b[2]	c[0]	c[1]	c[2]

atunci expresiile a[4], b[1], c[-2], sunt echivalente, în sensul că ele reprezintă adresa aceleiași zone de memorie.

6.3. Relația între pointeri şi tablouri

Să considerăm următoarea secvență:

```
int tab[10];
int *pa;
pa = &tab[0];
```

Ultima instrucțiune asignează variabilei pointer pa adresa primului element al tabloului tab. Notația "*pa" este echivalentă cu notația tab[0] iar notația "*(pa+1)" este echivalentă cu notația tab[1]. Așadar, în general "*(pa+i)" este totuna cu tab[i]. Următoarele secvențe for sunt echivalente:

```
for ( i = 0; i < 10; i++)
tab[i] = 100;
```

```
for ( i = 0, pa = &tab[0]; i < 10; i++)
*(pa+i) = 100;
```

```
for ( i = 0, pa = &tab[0]; i < 10; i++)
*(pa++) = 100;
```

În limbajul C, numele unui tablou poate fi utilizat ca pointer (constant) la adresa de început a tabloului. De aceea, cu declarațiile de mai sus, este perfect validă asignarea

```
pa = tab;
```

care este echivalentă cu "pa=&tab[0];". Nu sunt valide (e lesne de dedus) expresiile:

```
tab = pa;        sau        tab ++;
```

Așadar se poate scrie și:

```
for ( i = 0, pa = tab; i < 10; i++, pa++)
      *pa = 100;
```

În rezumat, echivalența între elementele unui tablou și numele său se exprimă prin:

tablou[i]	*(*tablou* + i)
&*tablou*[i]	*tablou* + i

6.4. Operații aritmetice cu pointeri

Operațiile aritmetice permise asupra pointerilor sunt adunarea/ scăderea unei constante, incrementarea/decrementarea (++, --) și scăderea a doi pointeri de același tip.

Trebuie precizat însă că unitatea care intră în discuție nu este octetul ci adresa obiectului pointat. Să urmărim următorul exemplu:

```
float    ftablou[10];
float *pf = ftablou;
pf++;
```

În acest exemplu, după incrementare, pf pointează la cel de-al doilea element al tabloului ftablou. Iată și alte exemple: în partea stângă se descrie o expresie în care apar pointeri iar în dreapta expresia echivalentă în termenii tabloului.

Notația pointer	Notația tablou
ptr = tablou	i = 0
ptr1 = ptr + 5	i = i + 5
ptr + i	tablou [i]
ptr + 4	tablou [4]
ptr - 3	tablou [-3]
ptr1 - ptr	j - i
ptr ++	i ++

6.5. Şiruri de caractere

În limbajul C, nu există un tip de bază pentru șiruri (lanțuri) de caractere. Pentru a reprezenta un șir de caractere, se utilizează tablourile de caractere. Prin convenție, ultimul caracter dintr-un astfel de lanț este caracterul NULL (' \0').

Tabloul de caractere (pe care-l numim mesaj):

's'	'a'	'l'	'u'	't'	'!'	' \o'

poate fi declarat prin una din următoarele trei instrucțiuni:

```
char mesaj[7] = {'s','a','l','u','t','!', '\o'};
char mesaj[7] = "salut!";
char mesaj[ ] = "salut!";
```

Citirea respectiv tipărirea acestui șir se face prin instrucțiunile:

```
scanf("%s", mesaj);
printf("%s", mesaj);
scanf("%s", &mesaj[0]);
printf("%s", mesaj[0]);
```

Există o bibliotecă de funcții standard pentru manipularea șirurilor de caractere care se prezintă în anexă.

6.6. Tablouri cu mai multe dimensiuni

Declararea unui tablou cu N dimensiuni se face prin:

tip nume[*dim_1*][*dim_2*]...[*dim_n*];

În această declarație, *dim_1, dim_2, ..., dim_n* trebuie să fie constante. Un tablou declarat în acest mod se compune din *dim_1*dim_2*...*dim_n* elemente de același tip (declarat) stocate într-o zonă contiguă de memorie.

Pentru a accesa un element dintr-un tablou exist\ mai multe moduri de notare, `n concordan]\ cu modul de stocare a elementelor tabloului `n memorie.

Iată de exemplu o declarație a unui tablou cu două dimensiuni de valori 3 respectiv 4 (așadar o matrice cu 3 linii și 4 coloane).

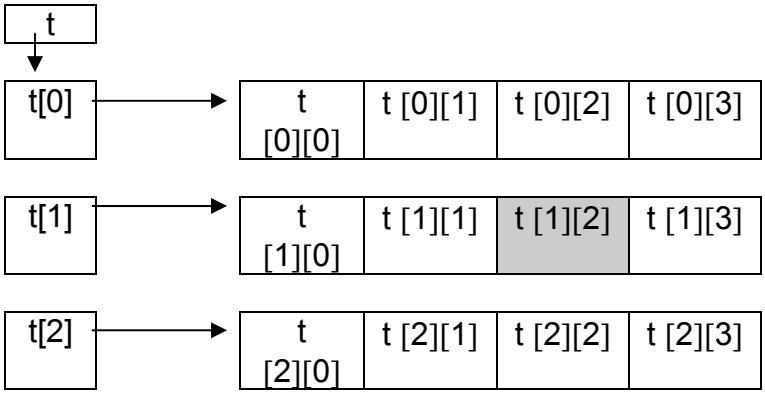
```
int t[3][4];
```

O prim\ viziune a reprezent\rii acestuia `n memorie este cea natural\ a unei matrice:

t [0][0]	t [0][1]	t [0][2]	t [0][3]
t [1][0]	t [1][1]	t [1][2]	t [1][3]
t [2][0]	t [2][1]	t [2][2]	t [2][3]

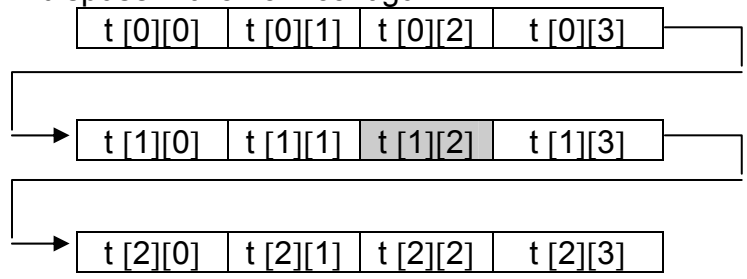
Elementeul hașurat din acest tablou se referențiază prin t[1][2] (este elementul de pe linia 2 și coloana 3 din matrice, datorită numerotării indicilor de la 0).

O altă viziune (acceptată de C) a acestui tablou este aceea a unui tablou cu o singură dimensiune în care fiecare element este la rândul său un tablou cu o dimensiune:



Elementul t[1] reprezintă un tablou cu o dimensiune iar elementul hașurat este referențiat prin *(t[1]+3).

~n sf=r[it a treia viziune a tabloului este aceea a unei succesiuni de elemente dispuse `ntr-o zon\ contigu\:



Elementul hașurat (același ca și în cazurile anterioare) se referențiază prin:

```
*( *t + 6)
```

sau prin notația echivalentă:

```
*( &( t[0][0] + 6))
```

Să notăm că în acest caz t este de tip pointer de pointer de întregi (dublă adresare indirectă).

O problemă importantă relativ la tablourile cu mai multe dimensiuni este cea a inițializării acestora. Așa cum referențierea unui element al tabloului poate fi făcută în mai multe moduri, și inițializarea unui tablou se poate face în moduri diferite (sintactic vorbind). Să exemplificăm aceste moduri cu referiri la același tablou pe care l-am reprezentat mai sus, t[3][4]:

- ini]ializarea prin enumerarea tuturor elementelor tabloului (`n ordinea liniilor dac\-i vorba de matrice):

```
int t[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

- inițializarea fiecărei dimensiuni de tablou, separat:

```
int        t[3][4] ={{1, 2, 3, 4},{5, 6, 7, 8},
```

```
        {9, 10, 11, 12}
    };
```

- inițializarea numai a unor elemente din tablou:

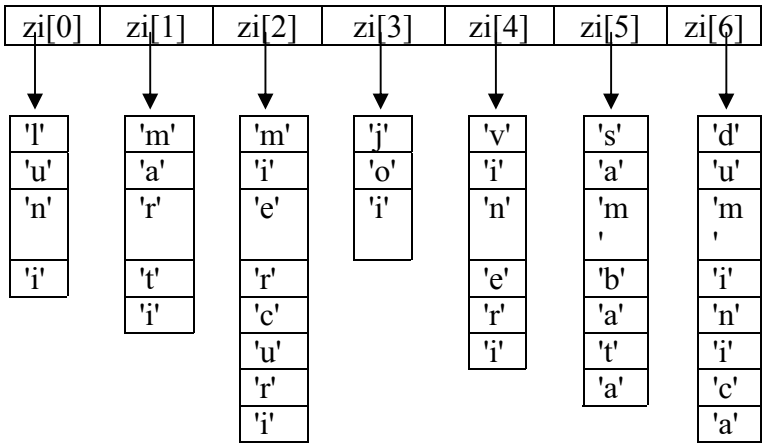
```
int     t[3][4] = {{1},{5, 6}};
```

Astfel t [0][0] va fi inițializat cu 1, t [1][0] cu 5 iar t [1][1] cu 6; restul elementelor sunt inițializate cu zero.

6.7. Tablouri de pointeri

Pointerii fiind variabile, pot intra în componența unui tablou. Un tablou unidimensional de pointeri corespunde la un tablou (cu două dimensiuni) de elemente de tipul pointat. Iată cum se declară un tablou de pointeri cu 7 componente ce se inițializează cu adresele șirurilor constante ce reprezintă numele zilelor dintr-o săptămână.

```
char *zi[7] = {"luni", "marti", "miercuri",
        "joi", "vineri", "sambata", "duminica");
    };
```



Să considerăm un tablou de pointeri de tip char, cu numele tabp. O funcție care să afișeze toate șirurile de caractere (pointate de tabp) la un singur apel ar putea arăta astfel:

```
void scrie(char *tabp [ ], int n)
{
        int i;
        for (i = 0; i < n; i++)
                if (tabp [i] != NULL)
                        printf ( "%s\n", tabp[i] );
}
```

Apelul acestei funcții pentru scrierea zilelor săptămânii este:

```
        scrie (zi, 7);
```

Să mai dăm un exemplu de funcție care sortează lexicografic (alfabetic) șirurile pointate de tabp:

```
void sortare (char* tabp[ ], int n){
        int i, sortat = 0;
        char *temp;
        while (! sortat) {
                sortat = 1;
                for (i = 0; i < n-1; i++)
                        if (strcmp (tabp [i], tabp [i+1]) > 0) {
```

```
                        temp =tabp [i];
                        tabp [i] = tabp [i+1];
                        tabp [i+1] = temp;
                        sortat = 0;
                    }
    }
}
```

Apelul pentru sortarea alfabetică a zilelor săptămânii este

```
sortare(zi, 7);
```

În corpul funcției sortare s-a folosit un apel la funcția strcmp care compară 2 șiruri de caractere s_1 și s_2; rezultatul este negativ dacă $s_1 < s_2$, pozitiv dacă $s_1 > s_2$ și zero dacă $s_1 = s_2$.

6.8. Pointeri şi alocarea memoriei

Printre funcțiile de alocare a memoriei (vezi Anexa), "maloc" și "free" joacă un rol important pentru că ele permit alocarea / dealocarea în mod dinamic a spațiului din memorie. Acest spațiu este posibil de accesat cu ajutorul variabilelor de tip pointer.

```
# include <stdlib.h>

void *malloc (size_t  dim);
void free(void *ptr);
```

Alocarea dinamică a memoriei este deosebit de utilă atâta timp cât programatorul folosește structuri de date pentru care nu știe dimensiunea la scrierea programului. În cursul execuției programul poate aloca memorie în limitele impuse de sistem.

Funcția malloc are ca argument numărul de octeți doriți a fi alocați și întoarce ca rezultat un pointer către zona de memorie liberă alocată de sistem. Pointerul returnat este de tip "void" ce corespunde unui pointer la o variabilă de tip oarecare. Pentru specificarea mărimii unui tip particular, se poate utiliza operatorul "sizeof" (vezi secțiunea următoare). Dacă nu mai este memorie disponibilă (nu se poate face alocarea) funcția malloc returnează valoarea NULL.

Iată un exemplu:

```
char *tampon;
element *ptr;
/*presupunem ca tipul element este definit undeva */
--------
tampon = malloc(512);
    /* s-a alocat o memorie tampon de 512 caractere*/
ptr = malloc(32*sizeof(element));
/*s-a alocat o zona de 32 de "elemente"*/
if ((tampon == NULL || ptr == NULL)) {
        printf ("Alocare imposibila ! \n");
                ...
...
```

Funcția "free" permite dealocarea spațiului de memorie alocat în prealabil de malloc. De pildă, după utilizarea memoriei tampon și ptr (în exemplul de mai sus) se scrie (în program)

```
free (tampon);
free (((char*)ptr);
```

6.9. Operatorul "sizeof"

```
sizeof expresie;
sizeof(tip);
```

Operatorul "sizeof" oferă dimensiunea în octeți a tipului expresiei considerate(tipului respectiv). Dacă se aplică unui tablou, operatorul sizeof oferă dimensiunea în octeți a memoriei ocupată de întreg tabloul. Pentru a obține numărul elementelor tabloului t se utilizează expresia "sizeof t" sau "sizeof t[0]". Să notăm în același timp că, atunci când este parametrul unei funcții, un tablou este considerat ca și un pointer; prin urmare operatorul sizeof nu oferă dimensiunea memoriei ocupată de tablou ci dimensiunea tipului pointer. Acest lucru se ilustrează în exemplul următor:

```
double tab [100];
void f(double[ ]);
/*declaratia este echivalenta cu void f(double*)*/
int main( ){
int i = sizeof tab;
/* i este initializat cu 100 * sizeof(double)*/
f(tab);
...
}

void f(double t[ ]){
int i = sizeof t;
/* i este initializat cu sizeof(double)*/
}
```

Pentru a obține dimensiunea memoriei ocupată de un tip anume se folosește:
sizeof(*tip*);
De aceast\ dat\ parantezele sunt obligatorii.
Apelul sizeof(long); oferă numărul de octeți utilizați pentru a memora un întreg "long" (este incorect a scrie sizeof long).

6.10. Pointeri către funcții

O funcție are un nume care este identificator dar acesta nu constituie o variabilă în C. Este posibil, pe de altă parte, de a defini o variabilă de tip pointer către o funcție. Declarația unui pointer la o funcție este de forma:

```
tip(*nume)(lista_arg);
```

tip: este tipul returnat de funcția pointată;
nume: este identificatorul pointerului la funcție;
lista_arg : este lista argumentelor funcției pointate.
De exemplu declarația:
char *(*p)(char*,const char*);
precizează că p este un pointer la o funcție de tip char* (pointer la tipul char), are doi parametri, unul de tip char*, celălalt de tipul const char*.

Este important a se pune "*nume" în paranteză deoarece declarația "*tip* **nume* (*lista_parametri*)" precizeaază că *nume* este o funcție de tip "*tip**".

Un exemplu de utilizare a tipului pointer la o funcție este funcția de comparare "cmp" ce apare ca argument la funcția de sortare "qsort" (vezi anexa):

```
void qsort(const void* baza, size_t n, size_n t,
        int(*cmp)(const void* val1, const void* val2))
```

Funcția qsort este o funcție de sortare generală care permite sortarea unui tablou de elemente de tip oarecare(baza) prin specificarea unei funcții de comparare. Iată un program ce utilizează funcția qsort :

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

    enum {n_max = 10};
    int tab_int[n_max];
/* tab_int este tabloul ce contine lungimile liniilor **ce trebuie sortat */
    char* tab_nume[n_max] = {
        "Ionescu Ion",
        "Constantin Vasile",
        "Ion Gheorghe",
        "Constantin Mircea",
        "Constantinescu Ioana",
        "Vasiliu Catalina",
        "Vasilescu Maria",
        "Constantiniu Calin",
        "Vasile Radu",
        "Vasilescu Cristi" };
/* tab_nume este tabloul ce trebuie ordonat **lexicografic */
/* se definesc functiile de comparatie */
    int comp_int (const  void* n1, const void* n2)
    {
        return(*(int*)n1- *(int*)n2);
    }
    int comp_nume (const void* p1, const void* p2)
    {
        return(strcmp(*(char**)p1,*(char**)p2));
    }
    int main(void)
    {
      int i;
      printf("Lista initiala de nume : \n");
      for(i = 0; i < n_max; i++)
                printf("%2d: %s\n",i,tab_nume[i]);
      for (i = 0, i < n_max; i++)
                tab_int[i] = strlen(tab_nume[i]);
      qsort(tab_int, n_max, sizeof(int), comp_int);
      qsort(tab_nume,n_max,sizeof(char*),comp_nume);
      printf("Lungimile numelor sortate:\n");
      for(i = 0; i < n_max ; i++)
                printf("%2d: %d\n",i,tab_int[i]);
      printf("Nume ordonate lexicografic: \n");
      for(i = 0; i < n_max; i++)
                printf("%2d: %s\n",i,tab_nume[i]);
        return 0;
  }
```

6.11. Exerciții

1. Așa cum s-a mai precizat, în C, șirurile de caractere sunt memorate în tablouri în care ultimul element este un delimitator de sfârșit de șir: caracterul NULL('\0'). În anexă sunt date funcțiile din biblioteca standard care se aplică șirurilor de caractere. Să se scrie programe C care să implementeze funcțiile:

a) int strlen(const char*s) : funcție ce returnează dimensiunea șirului s;
b) char* strcpy(char* dest,const char* sursa) : funcție ce copie șirul sursa în șirul dest și returnează dest;
c) char* strcat(char* dest,const char* sursa): funcție ce concatenează șirul sursa la sfârșitul șirului dest și returnează valoarea lui dest;
d) int strcmp(const char* s1, const char* s2) : funcție ce compară șirurile s1 și s2 conform ordinii lexicografice. Dacă are loc relația s1 < s2, se returnează o valoare negativă, dacă s1 = s2 se returnează 0 și dacă s1 > s2 se returnează o valoare pozitivă.

6.12. Soluții

1. Implementarea funcțiilor de la a) la d) se poate face folosind notația tablou sau aritmetica pointerilor.

```
int strlen(const char *s){
        int i = 0;
        while(s[i++] != '\0')
        ;
        return  -- i ;
}

char *strcpy(char* dest, const char *sursa){
        int i = 0;
        while(sursa[i] != '\0') {
                dest[i] = sursa[i];
                i++
        }
                dest[i] = '\0';
        return dest;
}

char *strcat(char *dest, const char *sursa){
        while(*dest != '\0')
                dest++;
        while(*scr != '\0')
                *dest++ = *src++;
        *dest = '\0';
        return dest;
    }
char *strcmp(const char *s1, const char *s2){
        int i = 0;
        while((s1[i] == s2[i] && (s1[i]  != '\0'))
                i++;
        return s1[i] - s2[i];
}
```

Iată și un program care testează aceste funcții. Se folosește funcția assert (anexa) care are ca parametru o expresie logică ce descrie o proprietate ce trebuie verificată. Dacă evaluarea acestui parametru este valoarea fals (0) atunci assert returnează un mesaj de eroare.

```
# include <stdlib.h>
# include <assert.h>

int strlen(const char *s)
char* strcopy(char *dest, const char *sursa);
char* strcat(char *dest, const char *sursa);
int strcmp(const char *s1, const char *s2);
int main(){
    char *inceput = "Iat\";
    char *mijloc = "un mic";
    char *sfarsit = "exemplu ! ";
    char *mesaj;
    int lungime;
    lungime = strlen(inceput)+ strlen(mijloc)
                +strlen(sfarsit);
    assert(lungime==strlen("Iata un mic exemplu ! "));
    mesaj = malloc(lung + 1);
    strcat(strcat(strcopy(mesaj,inceput),mijloc),
                        sfarsit);
    assert(strcmp(mesaj, "Iata un mic exemplu !")== 0);
    assert(strcmp(mesaj,inceput) > 0);
    assert(strcmp(mesaj, "Iat un micut exemplu!") < 0);
    return 0;
}
```

Cap 7 CONSTRUIREA DE NOI TIPURI

7.1 Redenumirea unui tip
7.2 Tipul structură
7.3 Accesul și inițializarea câmpurilor unei structuri
7.4 Structuri autoreferențiate
7.5 Tipul enumerare
7.6 Uniuni
7.7 Exerciții
7.8 Soluții

În capitolul al doilea au fost introduse tipurile de bază ale limbajului C. În anumite programe este util să regrupăm diferite variabile pentru a exprima o anume relație ce le caracterizează. De pildă un tablou permite a regrupa o mulțime de elemente de același tip. În acest capitol vom vedea că în C putem construi "structuri" sau "uniuni" care permit regruparea elementelor de tipuri diferite în scopul de a defini un nou tip. De asemenea există posibilitatea de a renumi (redenumi) un anume tip.

7.1. Redenumirea unui tip

Instrucțiunea typedef permite redenumirea unui tip. Sintaxa ei este:

```
typedef tip nume_nou;
```

Aici "*tip*" este tipul ce se dorește a fi redenumit iar *nume_nou* este numele ce se dă acestuia. Instrucțiunea permite redenumirea atât a tipurilor de bază (deja existente) cât și a tipurilor construite.

Iată câteva exemple:

```
typedef long Mare;
typedef char Mesaj[49];
typedef char Luna[10];
Mare a,b;
Mesaj salut="!Bine a-ti venit la cursul<Limbajul C>";
Luna ianuarie = "ianuarie";
```

7.2. Tipul structură

Instrucțiunea pentru definirea structurilor este:

```
struct nume{
    declarații
};
```

Structura este o colecție de una sau mai multe variabile (numite membri) grupate sub un singur nume. În instrucțiunea de mai sus "*nume*" este facultativ; el este totuși util pentru a identifica structura definită. Fiecare linie din partea "*declarații*" definește un câmp al structurii. O variabilă de tipul " struct *nume*" poate fi definită în aceeași instrucțiune în care este definită structura sau separat Iată câteva exemple:

```
struct Persoana{
        char nume[20];
        char prenume[20];
        int varsta;
} amicul;

struct Persoana colegul;

struct Data{
        short ziua;
        short luna;
        short anul;
};
struct Student{
        char *nume;
        char *prenume;
        struct Data data_nasterii;
        int an;
        char *grupa;
}
struct Student ionescu, popescu;
typedef struct{
        short ziua;
        short luna;
        short anul;
} Data;

typedef struct {
        char nume[20];
        char prenume[20]
        Data data_nasterii;
} Persoana;

Persoana ionescu, popescu;
```

Să observăm că structurile se pot defini fără nume; prin typedef li se atribuie un nume.

7.3. Accesul şi inițializarea câmpurilor unei structuri

Accesul la câmpurile unei variabile de tip structură se face utilizând numele unei variabilei urmată de punct (“.”) și de numele câmpului selecționat. Elementul desemnat în acest mod este echivalent cu o variabilă de același tip cu câmpul definit în structură. Iată,de pildă, urmărind exemplele din paragraful precedent, avem:

ionescu.nume este de tip char*;
ionescu.data_nasterii.ziua este de tip short;
amicul.varsta este de tip int.

Să considerăm acum declarația:

```
typedef struct{
        char nume[40] ;
        char prenume[40];
        int varsta;
} Persoana;
Persoana ionescu, dan, *necunoscut;
```

Aici, variabilele “ionescu” și “dan” sunt definite ca fiind de tip Persoana , care este un tip structură, iar necunoscut este o variabilă de tip pointer la structura Persoana . Accesul la câmpurile unei astfel de variabile(pointer la o structură) se face adăugând după numele varibilei pointer o “săgeată” (“->”) , urmată de numele câmpului.

Exemplu:

```
necunoscut = &ionescu;
necunoscut ->varsta = 41;
dan.nume = necunoscut->nume;
```

O formă echivalentă este următoarea:

```
(*necunoscut).varsta = 41;
dan.varsta = (*necunoscut).nume;
```

Inițializarea unei variabile de tip structură se poate face în maniera în care se inițializează un tablou: se descrie cu ajutorul unei liste de valori inițiale câmpurile structurii:

```
Persoana ionescu = { "Ionescu", "Vasile" , 42 };
Persoana necunoscut = { " " , " " , 0};
```

Două structuri de același fel pot să apară într-o expresie de atribuire. O atribuire de structuri este o copiere bit cu bit a elementelor corespunzătoare, încât o astfel de expresie are sens doar dacă structurile sunt de același tip. Are sens asfel:

```
necunoscut  = ionescu ;
```

Prin aceasta, câmpurile din variabila “ionescu” sunt copiate în câmpurile corespunzătoare din variabila “necunoscut”.

7.4. Structuri autoreferențiate

Utilizarea structurilor permite construirea de structuri de date evoluate ca: fișiere, liste, arbori etc.. De pildă, într-un arbore binar, fiecare nod este compus dintr-o informație (valoarea nodului) și doi pointeri, unul la subarborele stâng altul la cel drept. Iată cum poate fi definită o astfel de structură:

```
typedef struct {
...
} valoare;
```

```
/* s-a definit tipul "valoare" pentru **informatia din nodurile arborelui*/
typedef struct Arbore{
        valoare val;
        struct Arbore *stang;
        struct Arbore *drept;
} Arbore;
```

Să observăm că structura Arbore se autoreferențiază: printre câmpurile structurii arbore sunt două care sunt de acelați tip struct Arbore. Din acest motiv structura are un nume: Arbore. Programul următor ilustrează utilizarea unei structuri autoreferențiate: se construiesc doi arbori binari folosind funcția "adauga" și se afișează un arbore în notația infix (funcția "afisare").

Programul Arbori binari.

```
# include <stdio.h>
# include <stlib.h>
typedef int valoare;
typedef struct Arbore{
        valoare val;
        struct Arbore* stang;
        struct Arbore* drept;
} Arbore;
Arbore* adauga(valoare v, Arbore* s, Arbore* d);
void afisare (Arbore* a);
int main( ){
        Arbore* a1;
        Arbore* a2;
        a1=adauga(1, adauga(2, adauga(3, NULL, NULL),
    NULL), adauga(4, NULL, NULL));
        printf("Primul arbore este: \n");
        afisare(a1);
        a2=adauga(10, a1, adauga(20, NULL, NULL));
        printf("\n");
        printf("Al doilea arbore este: \n");
        afisare(a2);
        printf("\n");
        return 0;
}
    Arbore* adauga(valoare v, Arbore* s, Arbore* d)Ș
        Arbore* c;
        c=malloc (sizeof (Arbore));
        c->val = v;
        c->stang = s;
        c->drept = d;
        return c;
    }
    void afisare (Arbore* a){
        if (a! = NULL){
                printf ("(%d", a->val , " ");
                afisare (a->stang);
                printf (" ");
                afisare (a->drept);
```

```
            printf (")");
        } else
            printf ("nil");
    }
```

Execuția programului are ca rezultat afișarea următoarelor informații:

```
Primul arbore este:
(1 (2 (3 nil nil) nil) (4 nil nil))
Al doilea arbore este:
(10(1 (2 (3 nil nil) nil) (4 nil nil)) (20 nil nil))
```

7.5. Tipul enumerare

Definirea unui tip enumerare se face prin instrucțiunea:

enum *nume* { *lista_de_identificatori*};

Un tip enumerare permite ca, într-o maniera elegantă,să definim un tip care nu poate lua valori decât într-o mulțime precizată de valori constante. Primul identificator constant din lista de identificatori capătă valoarea 0, al doilea valoarea 1, etc. cu excepția cazului în care programatorul oferă valori inițiale acestor identificatori.

Exemple:

```
enum Bool {da, nu};
```

Aici da are valoarea 0 iar nu are valoarea 1.

```
enum Bool {da=1, nu=0};
enum Bool {nu, da}
```

În ambele situații, nu are valoarea 0 iar da are valoarea 1.

```
enum Luna {ian=1, feb, mar, april, mai,
iunie, iulie, august, sept, oct, nov, dec};
```

Aici, pentru că lui ian i s-a dat valoarea inițială 1, celelalte luni vor avea respectiv valorile 2, 3, ..., 12.

Similar cu definirea variabilelor de tip structura se definesc variabilele de tip enumerare:

```
enum Bool raspuns;
enum Bool spune=da;
```

Evident se poate redenumi și tipul enumerare:

```
Typedef enum {da, nu} Bool
Bool raspuns=nu;
```

7.6. Uniuni

Uniunile sunt structuri care pot conține (la momente de timp diferite), obiecte de tipuri diferite. De fapt, este vorba despre zone de memorie care, la un moment dat conțin un anume tip de variabilă iar la alt moment acest tip se poate schimba. Sintaxa este similară cu cea care definește o structură:

```
union nume {
    declarații
};
```

Lista de declarații, spre deosebire de cea de la structură, care definea câmpurile acesteia, definește o listă de alegeri posibile. Uniunea trebuie privită ca o structură în care membrii ocupă aceeași zonă de

memorie; dimensiunea memoriei alocată pentru o variabilă de tip uniune este dimensiunea celui mai mare membru al uniunii. Să notăm că, la un moment dat doar un membru al uniunii poate ocupa memoria.

În exemplul următor se specifică faptul că un punct în plan poate fi definit fie prin coordonatele sale carteziene fie prin cele polare.

```
Typedef union Ş
    struct {
            long abscisa;
            long ordonata;
    } cart;
    struct {
            float ro;
            float teta;
    } pol;
} Coordonate;
float pi=3.1415926;
Coordonate p1, p2;
...
printf("%d%d\n",p1.cart.abscisa,p1.cart.ordonata);
printf("%f%f\n",p2.pol.ro,p2.pol.teta);
```

O variabilă de tip Coordonate, cum sunt de pildă p1 și p2, poate conține fie valori întregi reprezentând coordonatele carteziene fie valori flotante reprezentând coordonatele polare. Desigur că în acest caz se presupune că prin program se află că p1 este de tip cart iar p2 este de tip pol. Declarațiile precedente pot fi modificate astfel (pentru a memora în variabilă însăși tipul de reprezentare ales) :

```
typedef enum {cart=1,pol=2,nec=0} TipCoord;
typedef union {
        TipCoord tip;
        struct {
                TipCoord tip;
                long abscisa;
                long ordonata;
        } cart;
        struct {
                TidCoord tip;
                float ro;
                float teta;
        } pol;
} Coordonate;
```

C=mpul “tip” este prezent `n toate alternativele uniunii [i el poate fi consultat prin referin]a p1.tip sau p1.cart.tip sau p1.pol.tip. Se mai poate elimina [i aceast\ redondan]\ prin urm\toarea declara]ie:

```
typedef struct {
        TipCoord tip;
        union {
                struct {
                        long abscisa;
                        long ordonata;
```

```
                } cart;
                struct {
                        float ro;
                        float teta;
                } pol;
        } val;
} Coordonate;
```

Dezavantajul este că numirea coordonatelor se face printr-o "cascadă" mai mare :

p1.val.cart.abscisa
p1.val.cart.ordonata , dacă p1.tip=1
p2.val.pol.ro
p2.val.pol.teta , dacă p2.tip=2

7.7. Exerciții

1. Scrieți un program pentru consultarea unei agende telefonice la nivelul personalului unei societăți. Pentru aceasta va trebui să se realizeze următoarele:

1.1 crearea agendei telefonice sub forma unui fișier text. Acest fișier va conține o mulțime de linii, fiecare având câte două câmpuri: un șir de caractere ce înseamnă identitatea abonatului și un număr care corespunde numărului său de telefon. Iată un exemplu:

andrei 1427
anca 2130
ana 2131
barbu 2140
bogdan 1430
… …
ștefan 2143
victor 2114
zeno 1442

1.2 crearea programului care realizează următoarele:

a) inițializarea unui tablou de elemente structurate prin citirea fișierului ce reprezintă agenda telefonică;
b) consultarea agendei: la furnizarea unui nume programul oferă numărul de telefon corespunzător (dacă acesta există).

De exemplu:

- anca
  tel.: 2130
- anuta
  persoană necunoscută
- ștefan
  tel.: 2143
- quit

2. Să se scrie un program care să realizeze:

a) alcătuirea unei liste de adrese care folosește un tablou de structură pentru stocarea informațiilor legate de adrese (nume, strada, oraș, codul poștal);
b) completarea tabloului de structuri cu noi adrese;
c) afișarea pe ecran a listei de adrese.

7.8. Soluții

1. Vom construi două funcții pentru căutarea în agenda telefonică: caută_secv pentru căutarea secvențială și caută_dih pentru căutarea dihotomică(binară).

```
# include <stdio.h>
# include <string.h>
# define dim_agenda 100
# define lung_nume 20
typedef char sir[lung_nume]
typedef struct {
    sir nume;
    int tel;
} persoana;
/* s-a definit o structura de tip persoana*/
persoana agenda[dim_agenda];
/* s-a definit agenda ca un tablou de persoane */
int n=0;        /* numarul persoanelor din agenda */
/* urmeaza functia de cautare secventiala */
int cauta_secv(char cineva[ ]) {
    int compar=1;
    int i=0;
    while ((compar>0) && (i<n)) {
        compar=strcmp(cineva, agenda[i].nume);
        i++;
    }
    return (compar==0)? i-1:-1;
}
/* urmeaza functia de cautare dihotomica */
int cauta_dih(char cineva[ ]){
    int compar=1;
    int i;
    int a=0;
    int b=n-1;
    while ((compar!=0) && (a<=b)) {
        i=(a+b)/2;
        compar=strcmp(cineva,agenda[i].nume);
        if (compar<0)
                b=i-1;/* se cauta in prima parte */
        else if (compar>0)
                a=i+1;/* se cauta in a doua parte */
    }
    return (compar==0)? i:-1;
}
int main(void){
    FILE* fisier;
    sir cineva;
    int i, fin=0;
    /* initializarea agendei telefonice */
    fisier=fopen("telefon.txt","r");
    do {
```

```
        fscanf(fisier,"%s%d",&(agenda[n].nume),  &(agenda[n].tel));
        n++;
    } while (!feof(fisier));
    printf("Consultare agenda telefonica\n");
    printf("\t Dati prenumele celui cautat \n");
    printf("\t sau  "quit" pentru sfarsit \n");
    while (!fin) {
        printf(">");
        scanf("%s", & cineva);
        fin=(strcmp(cineva, "quit")==0);
        if (!fin) {
          i=cauta_secv(cineva);
          /* sau
          i=cauta_dih(cineva);
          */
          if (i==-1)
             printf("tel.%s:necunoscut!\n",cineva);
          else      printf("tel.%s:%d\n",cineva,agenda[i].tel);
        }
    }
    return 0;
}
```

2. Programul conține funcțiile:

init_lista	-inițializarea listei;
select_meniu	-afișarea opțiunilor;
introducere	-adăugarea unui nou nume în următoarea structură liberă;
gaseste_liber	-explorează tabloul de structuri în căutarea unui element nefolosit;
sterge	-ștergerea unei adrese;
afiseză	-afișarea listei de adrese.

```
/* Program pentru listarea unor adrese folosind un **tablou de structuri*/

# include <stdio.h>
# include <stdlib.h>
# define MAX 100
struct adresa {
    char nume[30];
    char strada[40];
    unsigned long int numar;
    char oras[20];
    unsigned long int cod_postal;
} adresa_info[MAX];
void init_lista(void), introducere(void);
void sterge(void), afiseaza(void);
int select_meniu(void), gaseste_liber(void);
void main(void){
    char optiune;
    init_lista();
```

```
    for(;;) {
        optiune=select_meniu();
        switch(optiune) {
                case 1: introducere();
                        break;
                case 2: sterge();
                        break;
                case 3: afiseaza();
                        break;
                case 4: exit(0);
        }
    }
}
void init_lista(void){
    register int t;
    for(t=0;t<MAX;++t)adresa_info[t].nume[0]= °\0°;
}
int select_meniu(void){
    char s[80];
    int c;
    printf("1. Introduceti un nume \n");
    printf("2. Stergeti un nume \n");
    printf("3. Afisati fisierul \n");
    printf("4. Iesire din program \n");
    do {
        printf("\n Introduceti optiunea dvs.: ");
        gets(s);
        c=atoi(s);
    } while (c<0 || c>4);
        return c;
}
void introducere(void){
     int liber;
     char s[80];
     liber=gaseste_liber();
     if (liber==-1) {
         printf("\n Lista este completa. ");
         return;
     }
     printf("Introduceti numele: ");
     gets(adresa_info[liber].nume);
     printf("Introduceti numele strazii: ");
     gets(adresa_info[liber].strada);
     printf("Introduceti numarul: ");
     gets(s);
     adresa_info[liber].numar=strtoul(s, °\0°,10);
     printf("Introduceti numele orasului: ");
     gets(adresa_info[liber].oras);
     printf("Introduceti codul postal: ");
     gets(s);
     adresa_info[liber].cod_postal=
                        strtoul(s, °\0°,10);
```

```
}
gaseste_liber(void){
     register int t;
     for(t=0;adresa_info[t].nume[0]&&t<MAX;++t);
     if (t==MAX) return -1;
          /* Nu mai este loc in lista*/
     return t;
}
void sterge(void){
     register int liber;
     char s[80];
     printf("Introduceti nr. inregistrarii: ");
     gets(s);
     liber=atoi(s);
     if (liber>=0 & liber<MAX)
          adresa_info[liber].nume[0]= °\0°;
}
void afiseaza(void){
     register int t;
     for (t=0; t<MAX; ++t) {
          if (adresa_info[t].nume[0]) {
                    printf("%s\n",adresa_info[t].nume);
                    printf("%s\n",adresa_info[t].strada);
                    printf("%lu\n",adresa_info[t].numar);
                    printf("%s\n",adresa_info[t].oras);
                    printf("%lu\n",
                              adresa_info[t].cod_postal);
          }
     }
     printf("\n\n");
}
```

Cap 8 GESTIUNEA MEMORIEI

8.1 Precizarea unui mecanism de memorare
8.2 Gestiunea automată a memoriei
8.3 Gestionarea "register" a memoriei
8.4 Gestionarea statică a memoriei
8.5 Variabile globale
8.6 Declararea variabilelor externe
8.7 Gestiunea dinamică a memoriei
8.8 Exerciții
8.9 Soluții

Limbajul C oferă programatorului posibilitatea de a preciza mecanismul pe care-l dorește pentru memorarea unei variabile. Acest mecanism se referă la gestionarea memoriei în cursul execuției programului. Memorarea poate fi:

- statică, în segmentul datelor programului;
- automată, în stiva de execuție;
- dinamică, în memoria disponibilă;
- într-un registru al calculatorului.

În general programatorul nu trebuie să se gândească explicit la un mecanism anume de memorare. Compilatorul asociază în mod automat unei variabile un tip de memorare care corespunde locului unde s-a făcut declararea. În același timp însă, nu este lipsit de interes ca programatorul să cunoască particularitățile fiecărui tip de mecanism de memorare și de asemenea să cunoască modul în care compilatorul va administra variabilele pe care le declară în program.

8.1. Precizarea unui mecanism de memorare

Programatorul are posibilitatea să precizeze un mecanism de memorare prin instrucțiunea:

mem tip identificator;

unde: *mem* –este unul din cuvintele cheie: auto, register, static, extern;
tip – este cuvântul cheie ce definește tipul variabilei;
identificator – este numele variabilei.
Așadar, o variabilă este identificată printr-un nume (*identificator*), posedă un anume *tip* și este administrată de un anume mecanism de memorare.
Tipul variabilei va determina:

- dimensiunea zonei de memorie alocată pentru a reprezenta variabila;
- interpretarea conținutului variabilei atunci când ea este supusă unor instrucțiuni.

Categoria de memorare (*mem*) căreia-i aparține variabila va determina:

- *vizibilitatea* variabilei: identificarea părții din program capabilă să folosească acea variabilă;
- *durata de viață* a variabilei: perioada de timp în care variabila este accesibilă;
- *inițializarea* variabilei: conținutul implicit (dacă există unul) al variabilei la crearea sa.

Vom trece în revistă aceste mecanisme de memorare.

8.2. Gestiunea automată a memoriei

Variabilele cu gestiune automată corespund variabilelor alocate în stivele de execuție. Declararea se face prin:

auto *tip identificator* ;

și trebuie să fie totdeauna în interiorul unui bloc. Cuvântul cheie "auto" nu este obligatoriu; în lipsa acestuia, o variabilă definită într-un bloc va fi o variabilă cu gestiune automată. Durata de viață și vizibilitatea unei variabile "auto" vor fi limitate la interiorul blocului în care este definită, începând cu locul de declarare. Inițializarea unei variabile "auto" se va efectua la fiecare intrare în bloc; valoarea inițială, dacă nu este dată explicit, va fi nedefinită. Să mai notăm că limbajul C standard (ISO) permite inițializarea tablourilor și structurilor în cazul gestiunii automate a memoriei.
Exemple:

```
  auto float x;
auto float *p = &x;
char mesaj[ ] = "Mesaj initial";
/*cuvantul auto a fost omis;el este implicit */
/* x are o valoare nedefinita pe cand p este un
**pointer care are ca valoare adresa lui x */
```

8.3. Gestionarea "register" a memoriei

register *tip identificator* ;

Cuvântul cheie "register" permite programatorului să indice faptul că variabila ce se definește se memorează în unul din regiștrii calculatorului.

Cum numărul de regiștri este limitat și depinde de tipul de mașină pe care rulează programul, cerința aceasta este satisfăcută în mod real doar dacă este posibilă. Utilizarea acestei clase de memorare trebuie

văzută ca o directivă de optimizare dată compilatorului; acesta ține cont de ea atâta timp cât există regiștri disponibili în mașină.

Declarația "register" trebuie să fie plasată în interiorul unui bloc. Ea este efectivă doar pentru variabilele care pot fi codificate pe un cuvânt-mașină: caractere, întregi, pointeri. În plus, este imposibil a se obține adresa unei astfel de variabile.

Valoarea inițială a unei variabile registru este, implicit, nedefinită.

Exemple de utilizare a unor astfel de variabile sunt în soluția exercițiului 2 din capitolul precedent.

8.4. Gestionarea statică a memoriei

static *tip identificator* ;

Variabilele statice sunt alocate în segmentul de date al programului; durata lor de viață este timpul de execuție al programului însuși. Implicit sunt inițializate cu 0 la încărcarea programului, înainte de execuția funcției main. În cazul tablourilor sau structurilor, ansamblul câmpurilor va fi inițializat cu zero. Vizibilitatea unei astfel de variabile statice depinde de locul în care a fost declarată:

- în interiorul unui bloc: va fi vizibilă doar în blocul acela (la fel ca variabilele auto);
- în exteriorul unui bloc: va fi vizibilă începând cu definirea sa până la sfârșitul fișierului.

8.5. Variabile globale

O variabilă globală este o variabilă statică care poate fi referențiată în orice loc al programului. O astfel de variabilă se declară prin:

tip identificator ;

în exteriorul oricărui bloc; prin urmare nu se poate defini o variabilă globală în corpul unei funcții. Ea este definită începând cu declarația sa, până la sfârșitul fișierului sursă. Pentru a defini o variabilă globală în alt fișier, se folosește cuvântul cheie "extern". Este ușor de înțeles că, utilizarea unei variabile globale vine în contradicție cu principiile programării modulare. În același timp ele permit:

- evitarea transmiterii unor argumente la funcții apelate, oferindu-le posibilitatea să acceadă la aceste argumente, declarate în alte fișiere, și având în fișierul local specificarea extern. Atenție însă că o variabilă globală poate fi modificată indirect pe perioada apelului la o funcție.
- inițializarea structurilor și tablourilor fie în mod explicit indicând valori inițiale, fie implicit cu valori 0.

8.6. Declararea variabilelor externe

O variabilă externă poate fi declarată în orice punct al programului prin:

extern *tip identificator* ;

Această declarație permite declararea locală a unei variabile definită în alt fișier(cu același nume). Această facilitate este legată de faptul că limbajul C permite compilarea și legarea separată a diferitelor module de program: prin declararea unei variabile extern se comunică tuturor fișierelor variabilele globale necesare programului. Cuvântul rezervat extern indică compilatorului faptul că tipurile și numele variabilelor care urmează au fost declarate în altă parte și nu mai alocă din nou spațiu de memorie pentru acestea.

Exemple:

Fișier 1	Fișier 2	Fișier 2
int x,y;	extern int x, y;	extern int x,y;
char c;	extern char c;	extern char c;
main(void){	fun1(void){...x=...};	funct1(void)
...	fun2(void){...y=...};	{
}	...	x=..
...	..	
		}

În cazul declarării funcțiilor ce se definesc în alt modul (fișier), cuvântul extern este facultativ. Declarația:

```
extern void functie(char);
```

este echivalentă cu:

```
void functie (char);
```

În cazul funcțiilor important este să se descrie signatura sa, adică tipul rezultatului și al parametrilor de apel.

8.7. Gestiunea dinamică a memoriei

Gestiunea dinamică a memoriei permite crearea de noi variabile pe parcursul execuției programului. Această gestiune se efectuează în memoria disponibilă care se numește "grămadă" (heap). Durata de viață a unei variabile alocată dinamic este explicită:

-variabila este creată în urma execuției operației de alocare a memoriei (funcția "malloc" în C și operatorul "new" în C++);

-variabila dispare în urma execuției operației de restituire a memoriei (funcția "free" în C și operatorul "delete" în C++).

Să notăm că o variabilă alocată dinamic nu are un nume explicit. Ea este referențiată folosind o variabilă pointer (vezi cap. 6) care conține adresa sa. De aceea, vizibilitatea unei variabile alocată dinamic este legată de tipul de gestiune al pointerilor care referențiază indirect acea variabilă. De exemplu:

```
Persoana *ptr;
ptr=malloc(sizeof(Persoana));
```

Avem aici de-a face cu o variabilă alocată dinamic care este referențiată cu ajutorul pointerului ptr prin *ptr.

8.8. Exerciții

1. Să se construiască un program, format din trei module, care să implementeze funcțiile uzuale aplicate unei stive de caractere: push, pop, top, stiva_vidă, stiva_plină. Modulele vor fi:
   - -stiva.h, care conține declarațiile funcțiilor din modulul stiva.c;
   - -stiva.c, care conține implementarea funcțiilor;
   - -prog.c, care conține modulul principal.
2. Definiți un modul pentru gestionarea unei stive de șiruri de caractere. Caracteristicile modulului sunt definite în fișierul header "stiva.h" care are conținutul:

```
# define dimensiune_sir 80
# define dimensiune_stiva 10
typedef char sir[dimensiune_sir];
void push(const sir ch);
void pop(sir ch);
int stiva_vida(void);
int stiva_plina(void);
```

Utilizați funcțiile de manipulare a șirurilor de caractere (Anexa) pentru definirea funcțiilor acestui modul. Pornind de la acest modul scrieți un program test care realizează:

a) citirea unui șir de șiruri de caractere care se vor "împinge" în stivă (se vor stivui); citirea se va face până la întâlnirea sfârșitului de fișier sau până la umplerea stivei;
b) scoate șirurile din stivă și le afișează (evident în ordinea inversă stivuirii).

8.9. Soluții

1.
```
/* stiva.h : header ce contine declaratiile **functiilor din modulul stiva.c */
void push(char);
void pop(void);
char top_stiva(void);
int stiva_vida(void);
int stiva_plina(void);
/* stiva.c : modul de gestionare a stivei de **caractere */
# define dim 100;
static char stiva[dim];
static int index=0;
void push(char c){stiva[index++]=c;}
void pop(){--index;}
char top_stiva(){return(stiva[index-1]);}
int stiva_vida(){return(index==0);}
int stiva_plina(){return(index==dim);}
/* prog.c : modulul principal */
# include "stiva.h"
# include "stiva.c"
# include <stdlib.h>
# include <stdio.h>
int main()
{
    char element;
    printf("Incarcati stiva: \n>");
            while(!stiva_plina()&&scanf("%c",&element)!=EOF){
      push(element); printf(">");
    }
    printf("\n");
    printf("Iata continutul stivei: \n");
    while (!stiva_vida()) {
      element=top_stiva(); printf("<%c\n",element);
      pop();
    }
    return 0;
}
```

2.
```
 /*stiva.h*/
    # define dimensiune_sir 80
    # define dimensiune_stiva 10
    typedef char sir[dimensiune_sir];
    void push(const sir ch);
    void pop(sir ch);
    int stiva_vida(void);
    int stiva_plina(void);
```

/*modul.c

```
** Gestionarea unei stive de siruri de caractere.
*/
# include <stdio.h>
# include <assert.h>
# include <string.h>
# include "stiva.h"
/*
** variabile cu clasa de memorare *static*,
**nevizibile in alte fisiere.
*/
static int index=0;
/* indicile top-ului stivei */
static sir stiva[dimensiune_stiva];
void push(const sir c){
/* Se copie c in stiva dupa care se incrementeaza **index. Inainte de aceasta se apeleaza
functia **assert care impune  o conditie pentru a continua **executia: stiva sa nu fie plina.
*/
    assert(!stiva_plina());
    strcpy(stiva[index],c);
    index++;
}
void pop(sir c){
/* se decrementeaza index apoi se copie top-ul **stivei in c */
    assert(!stiva_vida());
    --index;
    strcpy(c,stiva[index]);
}
int stiva_vida(void){      return(index==0);}
int stiva_plina(void){
return(index==dimensiune_stiva);
}
/* test.c : programul de test ce consta din doua **iteratii: prima pentru citirea din fisierul de
**intrare si stivuirea informatiei, a doua pentru **afisarea rezultatului destivuirii.
*/
# include <stdio.h>
# include "stiva.h"
int main(void){
    sir x;
    printf("Se incarca stiva: \n>");
    while(!stiva_plina()&&scanf("%s",x)!=EOF) {
    push(x);
    printf(">");
    }
    printf("\n");
    printf("Iata continutul stivei: \n");
    while (!stiva_goala()) {
    pop(x);
    printf("<%s\n",x);
    }
    return 0;
}
```

Cap 9 CONVERSIA DE TIP

9.1 Mecanisme de conversie
9.1.1. Conversia la asignare
9.1.2. Evaluarea expresiilor aritmetice
9.1.3. Evaluarea expresiilor logice
9.1.4. Conversii explicite
9.1.5. Conversii de pointeri. Pointerul universal.

Conversia de tip în limbajul C oferă un mecanism simplu pentru evaluarea expresiilor aritmetice și pentru transmiterea parametrilor. Trebuie să facem distincția între două maniere de conversie: aceea care are drept scop schimbarea interpretării conținutului unei variabile și conversia care are drept consecință modificarea conținutului unei variabile. În primul caz de pildă, se poate interpreta conținutul unei variabile de tip caracter ca fiind o valoare întreagă și drept urmare posibilitatea utilizării acesteia într-o expresie aritmetică. În al doilea caz se poate converti o valoare reală la o valoare întreagă; ca o consecință se pierde valoarea fracționară a variabilei reale.

9.1. Mecanisme de conversie

Se disting patru cazuri `n care `ntr-un program C se aplică un mecanism de conversie:

1. asignarea cu tipuri diferite ale variabilelor termeni ai asignării:

```
float f;
int i=10;
f=i;
```

2. evaluarea unei expresii aritmetice sau logice în care componentele expresiei au tipuri diferite:

```
float pi=3.14;
int i=2;
.....
if (pi>2)
        pi=pi/i;
```

3. utilizarea operatorilor de conversie:

```
float f1=(float)14/5;    /* f1=2.8 */
```

```
float f2=14/5;          /* f2=2.0 */
```
apelul unei funcții:
```
printf("%d\n",'z');
/*caracterul 'z' este convertit la un intreg */
```

În cazul în care se utilizează operatori de conversie, compilatorul este cel care girează aceste conversii. Conversiile între tipurile de bază ale limbajului sunt considerate permise iar tratamentul la care sunt supuse datele de către compilator este bine stabilit de regulile limbajului. Alte conversii în care intervin tipuri definite de programator sunt considerate ilicite și sunt respinse de compilator. Vom detalia în continuare aceste mecanisme.

9.1.1 Conversia la asignare

În conversia la asignare, când apar două tipuri diferite, lucrurile se petrec după cum urmează:
1. asignări în care intervin două tipuri întregi:
   (a) reprezentarea tipului "sursă" este mai mare ca dimensiune decât
   reprezentarea tipului "destinatar":
   char=short/int/long
   short=int/long
   int=long

În acest caz biții de rang superior din valoarea din partea dreaptă a asignării sunt suprimați încât există, potențial, o pierdere de informație.

   (b) reprezentarea tipului "destinatar" este mai mare ca dimensiune
   decât cea a tipului "sursă":
   long=int/short/char
   int=short/char
   short=char

În acest caz nu există pierdere de informație: biții sursei sunt completați cu zero (păstrându-se eventualul bit semn). Dacă destinatarul este un tip fără semn, mecanismul este același; bitul semn în schimb nu se mai "transferă":
   unsigned long=int/short/char
   unsigned=short/char
   unsigned short=char

2. asignarea unei valori reale la o valoare întreagă:
   char/short/int/long=float/double

În acest caz sunt suprimate cifrele de la partea zecimală: este conversie prin trunchiere și nu prin rotunjire. Dacă valoarea părții întregi a sursei este prea mare pentru a fi reprezentată într-un întreg, apare o eroare (Floating point exception).
3. asignarea unei valori reale în dublă precizie la o variabilă reală în simplă precizie:
   float=double

În acest caz există o pierdere a preciziei reprezentării valorii reale. Poate apărea și aici eroarea "Floating point exception" dacă partea exponent este prea mare pentru o variabilă reală în simplă precizie.

9.1.2 Evaluarea expresiilor aritmetice

- tipurile întregi mai mici decât int sunt convertite la int;
- tipul float este convertit la double;

Faza II:

- se alege tipul de evaluare: întreg, real, fără semn, etc.

În rezumat, lucrurile se petrec astfel:

Dacă un operand este:	Celălalt operand se convertește la:	Rezultatul este:
double	double	double
unsigned long	unsigned long	unsigned long
long	long	long
unsigned	unsigned	unsigned

Exemple:

	i (int)	+	c (char)
Faza 1: Faza 2: Rezultatul:	int int	 int	int int

	i (int)	+	(a (long)	+	b) (float)
Faza 1 Faza 2 Rezultat Faza 1 Faza 2 Rezultat	 int double	 double	long double	 double double double	double double

9.1.3. Evaluarea expresiilor logice

Rezultatul evaluării unei expresii logice va fi 0 (reprezentând "false") sau 1 (reprezentând "true"). De aceea evaluarea unei expresii logice se face prin:

- evaluarea operanzilor: dacă valoarea aritmetică este 0 atunci aceasta se interpretează prin "false", în caz contrar prin "true";
- asignarea valorilor 0-1: dacă evaluarea expresiilor este "false" (true), valoarea asignată acesteia este 0 (1).

Exemple:

```
x=50;
b=(x>0) && (x<31);  /* b capata valoarea 0 */
i=10
while (i--) {
        .....
}
```

Înainte de prima evaluare a expresiei "i- -", i are valoarea 10 deci expresia capătă valoarea "true", după care, i se decrementează, etc.

9.1.4 Conversii explicite

Conversiile explicite se exprimă prin:

(*tip*) *expresie* sau *tip* (expresie_)

Atâta timp cât conversia este permisă, rezultatul expresiei este convertit la tipul indicat. Operanzii ce apar în această expresie rămân neschimbați.

Acești operatori de conversie permit programatorului să specifice o conversie care nu se face implicit de către compilator. Aceștia pot fi utilizați:

- pentru forțarea conversiilor în expresii;

- pentru transmiterea valorilor cu tipurile cerute ca parametri în funcții;
- schimbarea tipului valorii returnate de o funcție.

Exemple:

```
int a=21; b=4;
float f;
double r;
f=a/b;                  /* f5.0; */
f=(float)a/b;           /* f=5.25; */
f=a/(float)b;           /* f=5.25; */
....
r=sqrt(double(a));
....
a=(int)ceil(r);
/*functia ceil returneaza o valoare de tip intreg*/
```

9.1.5 Conversie de pointeri. Pointerul universal

Conversia între pointeri la tipuri diferite nu este implicită. Compilatorul forțează conversia și afișează un mesaj de avertisment ("warning") întotdeauna când într-un program apare o astfel de conversie. Să considerăm de pildă următoarea secvență:

```
char *alocare(int dim);
int *ptr;
ptr = alocare(100);
```

În cea de-a treia linie a secvenței de mai sus apare o asignare între pointeri incompatibili. Această incompatibilitate se poate evita în două moduri. O primă soluție constă în a utiliza o conversie explicită:

```
....
ptr=(int*) alocare(100);
```

O a doua soluție constă în a utiliza tipul pointer universal void*. O variabilă pointer de tip void* poate primi ca valoare pointer la un tip oarecare. Reciproc, o valoare de tip void* este implicit convertită la o valoare pointer de un tip oarecare. Pentru exemplul precedent se procedează astfel:

```
void* alocare(int dim);
int* ptr1
double* ptr2
ptr1=alocare(10);
ptr2=alocare(20);
```

Asignările pentru ptr1, ptr2 sunt corecte: tabloul de pointeri alocare este de tip void* încât acești pointeri pot fi asignați la pointeri de orice tip.

Tipul pointer universal void* este folosit frecvent în funcțiile disponibile din bibliotecile limbajului C (Anexa) în scopul de a realiza conversii implicite între pointeri de tipuri diferite.

Cap 10 PREPROCESORUL

10.1 Directive preprocesor
10.1.1 Constante simbolice. Directiva #define
10.1.2 Includerea fișierelor
10.1.3 Compilare condiționată
10.1.4 Directiva #error
10.2 Macroinstrucțiuni
10.2.1 Macroinstrucțiuni predefinite
10.2.2 Macroinstrucțiuni tip funcție

În codul sursă al unui program C se pot include instrucțiuni destinate compilatorului; acestea se numesc directive pentru preprocesor.

Preprocesorul realizează o fază de precompilare care are ca obiect:

- includerea unor fișiere (specificate) în program;
- compilări condiționate conform unor parametri (valori) transmise în comanda de compilare;
- definirea unor constante simbolice;
- definirea macroinstrucțiunilor. Programatorul asociază o secvență de instrucțiuni unui identificator. La apariția acestui identificator în program, preprocesorul substituie secvența corespunzătoare de instrucțiuni în locul acestui identificator.

Precompilarea este o fază de prelucrare independentă de compilarea propriu-zisă. În același timp însă, ea face parte integrantă din procesul de compilare: apelul compilatorului pentru un program execută mai întâi încărcarea preprocesorului. Rezultatul activității preprocesorului este un fișier sursă în care sunt tratate toate directivele de precompilare și care este transmis compilatorului.

10.1 Directive preprocesor

Preprocesorul tratează doar liniile ce încep prin caracterul "**#**"; acestea sunt directive către preprocesor. O directivă este de forma:

 # instrucțiune argumente

Principalele instrucțiuni pentru preprocesor sunt:

define	ifndef	undef
if	else	endif
elif	include	ifdif
line	error	pragma

De notat că un rând în fișierul sursă C nu poate conține mai mult de o directivă.

10.1.1 Constante simbolice. Directiva #define

O constantă simbolică asociază unui identificator un șir de caractere. Preprocesorul înlocuiește fiecare apariție a identificatorului, în textul sursă, prin șirul de caractere asociat. Constanta simbolică se definește cu directiva "define":

define *identificator șir_de_caractere*

Exemple:

```
# define NULL 0L
# define TRUE 1
# define FALSE 0
# define EOF (-1)
```

Utilizarea constantelor simbolice are diverse scopuri. Acest mecanism permite:

- a asigna unui identificator un șir de caractere;
- a defini existența unei constante simbolice:

= define ident

- a suprima definiția și existența unei constante simbolice:

```
# undef ident
```

Vizibilitatea constantelor simbolice începe cu definirea lor (directiva `#define`) și ține până la sfârșitul fișierului, cu excepția cazului când în acesta apare o directivă `#undefine` pentru identificatorul corespunzător.

Exemplu:

```
# define EOF (-1)
....
# define UNU
....
# undefine UNU
....
```

10.1.2 Includerea fişierelor

Directiva #include cere compilatorului să ia în considerare și un alt fișier sursă în afară de cel în care se află directiva. Forma acestei directive este:

include <*nume_fisier*> /* se caută fișierul începând cu directoarele standard */ sau # include "*nume_fisier*" /* se caută fișierul în directorul curent sau cel indicat prin cale */

Includerea fișierelor poate fi utilizată pentru:

- a insera fișiere sursă din biblioteci C care pot conține module de aplicații sau definiții de funcții;
- a insera fișiere "header" (care au extensia '.h'). Aceste fișiere conțin:
  1. constante simbolice și macroinstrucțiuni standard;
  2. declarații de funcții incluse într-o bibliotecă;
  3. definiții de tipuri.
  4.

Exemplu Pentru a folosi funcțiile din biblioteca matematică se include fișierul *math.h* care conține declararea acestor funcții:

```
# include <math.h>
....
val=pow(M_PI, (double)n);
/* se calculeaza puterea a n-a a lui π */
```

În fișierul math.h se află definiția constantei M_PI și a funcției pow( ).

10.1.3 Compilare condiționată

Directivele de compilare condiționată permit compilarea selectivă a unor porțiuni din programul sursă. Aceste directive sunt:

```
# if expr_constanta
# else
# endif
# elif expr_constanta
# ifdef identificator
# ifndef identificator
```

Directiva "if" prin "*expr_constantă"* care urmează după ea permite includerea porțiunii de cod între "if" și "endif" în procesul de compilare, doar dacă expresia este evaluată la "true" (în caz contrar această porțiune de cod este ignorată de compilator).

Directiva "else" oferă o alternativă în cazul în care "if" eșuează; este asemănătoare instrucțiunii if-else din limbajul C.

Directiva "elif" are sensul de "else if"; ea poate determina un lanț de if-else-if:

```
# if expresie
        cod;
= elif expresie1
        cod1;
= elif expresie2
        cod2;
....
# elif expresien
        codn;
= endif
```

Directivele "ifdef" (if defined) și "ifndef" (if not defined) sunt utilizate pentru compilarea condiționată de definirea anterioară a lui "*identificator*" într-o directivă de tipul #define. Ambele directive pot folosi o alternativă de forma #else dar nu și #elif.

În concluzie, compilarea condiționată este folosită pentru:

- scrierea de programe portabile: constantele simbolice ce se testează pot reprezenta tipul și caracteristicile mașinilor utilizate pentru o anume porțiune de cod;
- optimizarea unor coduri utilizând avantajele specifice fiecărei mașini;
- traseul de execuție al programelor în faza de test.

10.1.4 Directiva `#error`

Directiva #error cere compilatorului să sisteze compilarea. Are forma:

```
# error mesaj
```

La întâlnirea directivei este întreruptă compilarea și se afișează "*mesaj*"-ul ce este scris în linia directivei. Se utilizează la depanarea programelor.

10.2 Macroinstrucțiuni

Prin acest mecanism programatorul asociază o secvență de instrucțiuni unui identificator.

10.2.1 Macroinstrucțiuni predefinite

Limbajul standard ANSI C definește cinci nume de macroinstrucțiuni predefinite, încorporate în compilator. Acestea sunt:

LINE, _FILE_, _DATE_, _TIME_, _STDC_.

Identificatorul _LINE_ conține numărul liniei de cod la compilare iar _FILE_ este un șir care conține numele fișierului sursă compilat. Conținutul acestor doi identificatori poate fi modificat prin directiva #line:

```
# line număr;
# line "nume_fișier";
# line număr "nume_fișier";
```

Aici "*număr*" este orice întreg pozitiv; acesta devine noua valoare a lui _LINE_ în punctul în care se întâlnește directiva, iar "nume_fișier" este un identificator valid de fișier care devine noua valoare a lui _FILE_.

Macroinstrucțiunea _DATE_ conține un șir de forma "*lună/zi/an"* în care *lună, zi, an* sunt compuse din câte două cifre ce reprezintă în ansamblu data conversiei fișierului sursă în cod obiect. Ora la care a fost executată această conversie este conținută în _TIME_ sub forma șirului *ora/minute/secunde*.

Macroinstrucțiunea _STDC_ conține constanta 1 scrisă în baza 10. Aceasta semnifică faptul că implementarea este conform standardului ANSI C. Dacă _STDC_ nu-i definită sau conține altceva decât 1, implementarea nu este în variantă standard.

10.2.2 Macroinstrucțiuni tip funcție

Aceste macroinstrucțiuni utilizator se definesc cu directiva #define:

```
# define nume_macro
  secvență_caractere
```

Aici, "*nume_macro*" poate fi un identificator ce reprezintă numele macroinstrucțiunii. Acesta, utilizat într-un text sursă, va fi înlocuit cu "*secvență_caractere*" care poate însemna o secvență de instrucțiuni C (vezi și 10.1.1). Dar "*nume_macro*" poate fi și numele unei funcții urmată de parametri: prin aceasta se definește o macroinstrucțiune de tip funcție care poate fi 'apelată' în orice punct al programului în care a fost definită.

Exemplu:

```
# define <stdio.h>
# define ABS(a) (a)<0?-(a):(a)
void main(void){
    printf(″Valoarea absoluta a lui -1 si 1:
                %d%d″,ABS(-1),ABS(1));
}
```

Folosirea funcțiilor definite ca macroinstrucțiuni are, pe de o parte, un avantaj major: mărirea vitezei de execuție a codului fiindcă nu mai apare nici o încărcare suplimentară legată de apelarea funcției. Pe de altă parte însă, dacă dimensiunea macroinstrucțiunii tip funcție este mare, acest supliment de viteză este "anihilat" de creșterea dimensiunii programului, datorită codului care apare de mai multe ori.

Cap 11 EXERCIȚII

1. Să se scrie un program numit **maxmin.c** care să execute :
   a. citirea unui intreg pozitiv n;
   b. citirea a n numere reale;
   c. determinarea valorilor maxim și minim dintre cele n numere citite;

2. Să se scrie un program numit **sumedia.c** care să citească niste numere reale și, pentru fiecare să scrie un rând într-un tabel ce are forma:

Nr.crt.	Număr	Min	Max	Suma	Media
…	…	….	….	….	….

   unde Min(Max) înseamnă cel mai mic(mare) dintre numerele citite până în acel moment, Suma și Media fiind suma (media) numerelor citite până în acel moment

3. Să se scrie un program care să numere literele mici, literele mari , cifrele și celelalte caractere(la un loc) dintr-un text(fișier).
4. Să se scrie funcții recursive pentru calculul factorialului unui număr dat n, a puterilor 1,2 …, k a unui număr n, a sumei numerelor naturale m și n. Să se folosească aceste funcșii într-un program principal.
5. Să se scrie o funcție " int este_prim(int n) " care săz returneze 1 dacă numărul n este prim și zero în caz contrar. Să se scrie o funcție " fibonacci (int n) " care să returneze al n-ulea termen al șirului lui Fibonacci. Să se folosească cele două funcții pentru a verifica dacă al n-ulea termen din șirul lui Fibonacci este număr prim (n = 1, …, 12).
6. Folosind funcția "este_prim(…)" de la exercițiul precedent, scrieți un program ce să verifice conjectura lui Goldbach pentru toate numerele pare dintr-un interval dat [a,b] : Orice număr par n mai mare ca 2 este suma a două numere prime. De exemplu:
   700 = 17 + 683 , 702 = 11 + 691, 704 = 3 + 701, etc.
7. Următoarea funcție calculează, recursiv, cel mai mare divizor comun a două numere întregi pozitive p și q:

```
int cmmdc_r(int p, int q)
        {
                int r;
                if (( r = p % q ) == 0 )
                        return q;
                else
                        return cmmdc_r(q, r);
        }
```

   Scrieți un program pentru testarea acestei funcții precum și pentru calculul celui mai mare divizor comun a n numere date. Scrieți apoi și testați varianta iterativă a funcției, cu numele :
   int cmmdc_i (int p, int q)
8. Descrieți și implementați trei algoritmi de sortare.
9. Construiți o funcție " int numar_cuvinte(const char *s) " care să returneze numarul cuvintelor din șirul s (cuvintele sunt despărțite prin spațiu/spații). Să se testeze această funcție pentru un șir anume(fișier).

10. Să se scrie și apoi să se testeze funcții pentru adunarea a doi vectori și pentru adunarea a două matrice.
11. Să se scrie un program în care să se folosească funcția sistemului :

```
void qsort(void *array, size_t n_els,size_t el_size,
    int compare(const void *, const void *))
```

pentru a sorta o listă de numere reale în ordinea crescătoare relativ la părțile fracționare a acestor numere. De exemplu lista:
2.43, 45.01, 23.90, 123.04, 0.98
este sortată astfel:
45.01, 123.04, 2.43, 23.90, 0.98

12. Să se scrie funcțiile care implementează o stivă în C(push, pop, top, etc). Să se folosească acestea pentru a verifică dacă un șir de caractere format numai din literele a, b, c este palindrom cu centrul 'c':
șirurile: aabacabaa , bababbcbbabab, c, aca, bbcbb
sunt palindroame pe când șirurile:
aabbcbb. ccab, aaaaaaaacaaaabb, acb, aaaa
nu sunt palindroame.

13. Fie a un tablou bidimensional (m x n). Un punct sa al acestui tablou este un element a[i0,j0] cu proprietatea:
a[i0,j0] = min{a[i0,j] : 0<=j<=n-1} = max{a[i,j0] : 0<=i<=m-1}
Scrieti un program care determina punctele sa (daca exista) ale unui tablou bidimensional.
14. Se da o matrice patratica. Sa se ordoneze crescator fiecare linie a matricii, apoi sa se rearanjeze liniile astfel incat suma elementelor de pe diagonala sa fie minima.
15. Scrieti un program care, avand la intrare doua siruri de caractere, determina cel mai lung subsir comun si pozitiile la care acesta incepe in fiecare dintre cele doua siruri.
16. Proiectati structuri de date pentru reprezentarea unui punct, a unui triunghi, dreptunghi, cerc.

Scrieti proceduri de citire si scriere a unui punct, triunghi etc.

Scrieti o procedura care determina, pentru un triunghi dat, numarul punctelor de coordonate intregi aflate in interiorul sau.

Scrieti o procedura similara pentru cerc.

17. Proiectati o structura de date pentru reprezentarea datei calendaristice si scrieti subprograme care:

- verifica daca valoarea unei variabile din structura este o data valida.
- calculeaza data calendaristica care urmeaza unei anumite date.
- calculeaza data calendaristica care precede o anumita data.

18. Sa se proiecteze o structura de date pentru reprezentarea multimilor finite de numere complexe si sa se scrie proceduri care realizeaza operatii cu multimi de numere complexe reprezentate astfel.

BIBLIOGRAFIE

1. **Herbert Schildt** *C - Manual Complet*, Bucuresti, Ed. Teora 1998
2. **Liviu Negrescu** *Limbajele C si C++ pentru incepatori*, vol I- III, Cluj-Napoca

   Volumul I - *Limbajul C*

   Volumul II - *Limbajul C++*

   Volumul III - *Limbajele C si C++ in Aplicatii*
3. **Cristea,V.,C.Giumale, E.Kalisz. A.Pănsiu**, *Limbajul C standard.* Editura Teora, 1992.
4. **Grigoraș, Gh**. *Programarea calculatoarelor:Fundamente*, Editura "Spiru Haret", Iași, 1999.
5. **Pătruț, B.,** *Aplicații în C și C++,* Editura Teora, București, 1998.
6. **Muscă, Gh. și alții,** Informatică aplicată, manual pentru licee de informatică, cls.XI, Editura Didactică, București 1998

ANEXA

Biblioteca standard C

FIȘIERE HEADER			
<assert.h>	**<limits.h>**	**<signal.h>**	**<stdlib.h>**
<ctype.h>	**<locale.h>**	**<stdarg.h>**	**<string.h>**
<errno.h>	**<math.h>**	**<stddef.h>**	**<time.h>**
<float.h>	**<setjmp.h>**	**<stdio.h>**	

1. Fișierul <**assert.h**> conține macro-ul assert()
   - void assert (int expr);

   Dacă expr este zero, este tipărit un mesaj(diagnostic) și se iese din program.

2. Fișierul <**ctype.h**> conține macrouri (funcții) pentru testarea caracterelor:

- int isalnum(int c); /* c este alfanumeric? */
- int isalpha(int c); /* c este alfabetic? */
- int iscntrl(int c); /* c este car. de control? */
- int isdigit(int c); /* c este cifra? */
- int isgraph(int c); /* c este car. grafic? */
- int islower(int c); /* c este litera mica? */
- int isprint(int c); /* c este tiparibil? */
- int ispunct(int c); /* c este car. punctuatie? */
- int isspace(int c); /* c este spatiu? */
- int isupper(int c); /* c este litera mare? */
- int isxdigit(int c); /* c este cifra hexa? */

   Mai conține:
   int tolower (int c) și int toupper (int c)

   funcii ce permit transformarea literelor din mari țn mici sau invers.

3. Fișierul <**errno.h**> conține macrouri folosite pentru raportarea erorilor.

4. Fișirerul <**float.h**> conține definițiile unor constante pentru limitele unor valori flotante: DBL_MAX, FLT_MAX, DBL_MIN, DBL_EPSILON etc.

5. Fișierul <**limits.h**> conține definițiile unor constante ce caracterizează întregii: CHAR_BIT, CHAR_MAX, INT_MAX etc.

6. Fișierul <**locale.h**> conține construcții (structuri, funcții) ce permit setarea unor proprietăți "locale": forma punctului zecimal, informații monetare etc.

7. Fișierul <**math.h**> conține prototipuri pentru funcțiile matematice:

- double cos (double x); double acos (double x);
- double sin (double x); double asin (double x);
- double tan (double x); double atan (double x);
- double cosh (double x); double exp (double x);
- double sinh (double x); double log (double x);
- double tanh (double x); double log10 (double x);
- double ceil (double x); double flor (double x);
- double fabs (double x); double sqrt (double x);
- double atan2 (double y, double x);
- double fmod (double x, double y);
- double pow (double x, double y);

8. Fișierul <**setjmp**> conține declarații și prototipuri ce permite programatorului să utilizeze salturi nelocale.

9. Fișierul < **signal.h**> conține construcții ce permit programatorului să mânuiască condiții sau semnale excepționale.

10. Fișierul <**stdarg.h**> conține o redefinire (typedef) de tip și trei m macrouri ce permit scrierea de funcții cu un număr variabil de argument (ca și printf).

11. Fișierul <stddef.h> conține definiții de tip și macrouri ce se folosesc în alte definiții:

- typedef char wchar_t; /* tipul wide character*/
- typedef int ptrdiff_t; /*tipul diferență de pointeri */
- typedef unsigned size_t; /*tipul obținut din sizeof() */
- #define NULL;

12. Fișierul <**stdio.h**> conține macrourile, definițiile de tip și prototipurile funcțiilor utilizate de programator pentru a accesa fișiere.

Sunt definite aici constantele BUFSIZ, EOF, FILENAME_MAX, FOPEN_MAX, NULL, TMP_MAX. De asemenea sunt definiți pointerii stdin, stdout, stderr și pentru structura FILE. Funcții pentru accesul la fișiere:

- FILE *fopen (const char *filename, const char *mode);
- int fclose (FILE *fp);
- int fflush (FILE *fp);
- FILE *tmpfile (void);
- int fseek (FILE *fp, long offset, int place); /* Poziționează indicatorul pentru următoarea operație în fișier la "place" + "offset" */
- long ftell (FILE *fp); /* Returnează valoarea indicatorului de poziție a fișierului pointat de fp*/
- void rewind (FILE *fp); /* Setează indicatorul la începutul fișierului */
- int fsetpos (FILE *fp, const fpos_t *pos); /* Setează indicatorul la valoarea pos */
- int rename (const char *from, const char *to);

De asemnea fișierul mai conține prototipurile funcțiilor de intrare/ieșire (vezi cap.4).

13. Fișierul <**stdlib.h**> conține prototipurile funcțiilor de uz general : alocarea dinamică a memoriei, căutare binară, sortare, generare de numere aleatoare, comunicarea cu sistemul, conversie de stringuri,etc. Iată unele dintre ele:

- void *calloc(size_t n, size_t el_size);
- void *malloc(size_t size);
- void *realloc(void *ptr, size_t size);
- void free(void *ptr);
- void *bsearch(const void *key_ptr, const void *a_ptr, size_t n_els, size_t el_size, int compare(const void *, const void *));
- void qsort(void *a_ptr, size_t n_els, size_t el_size,
  int compare(const void *, const void *));
- int rand(void);
- void srand(unsigned seed);
- int abs(int i);
- long labs(long i);
- div_t div(int numer, int denom);
- ldiv_t ldiv(long numer, long denumer);
- double atof(const char *s);
- int atoi(const char *s);
- long atol(const char *s);
- double strtod(const char *s);
- long strtol(const char *s, char **end_ptr, int base);
- void abort(void);
- void exit(int status);

14. Fișierul <**string.h**> conține prototipuri de funcții pentru manipularea blocurilor de memorie sau a șirurilor:

- void memcmp(const void *p, const void *q, size_t n);
- void *memcpy(void *to, void *from, size_t n);
- void *memmove(void *to, void *from, size_t n);
- void *memset(void *p, int c, size_t n);
- char *strcat(char *s1, const char *s2); /*concatenare*/
- char *strchr(const char *s, int c); /*cauta c în șirul s*/
- int strcmp(const char *s1, const char *s2); /* comparare șiruri*/
- char *strcpy(char *s1, const char *s2) /* copie s2 în s1 */
- size_strlen(const char *s); /*lungimea șirului s */
- char *strncat(char *s1, const char *s2, size_t n); /*concatenarea a cel mult n caractere din s1 la s2*/
- int strncmp(const char *s1, const char *s2, size_t n); /* comparare a cel mult n caractere din șirurile s1 și s2*/
- char *strncpy(char *s1, const char *s2, size_t n) /* copie cel mult n caractere din s2 în s1 */
- char *strstr(const char *s1, const char *s2); /* caută în s1 prima apariție a lui s2 */
- char *strtok(char *s1, const char *s2); /* caută tokenuri în s1 folosind caracterele din s2 ca separatori */

15. Fișierul <**time.h**> conține prototipuri ale funcțiilor pentru obținerea datei, orei ți cele privind ceasul intern al calculatorului. Se folosește structura tm care este definită astfel:

```
struct tm {
        int tm_sec;             /*secunde 0..60*/
        int tm_min;             /*minutr 0..59 */
        int tm_hour;   /*ore 0..23 */
        int tm_mday; /*ziua lunii 1..31 */
        int tm_mon;             /*luna anului 0..11 */
        int tm_year;   /*anul de la 1900 */
        int tm_wday;  /* zilele de la 0 la 6 */
        int tm_yday;  /*zilele din an 0..365 */
        int tm_isdst;  /* Indicator de timp */
};
```

Funcții:

- typedef long clock_t;
- typedef long time_t;
- clock_t clock(void); /*returnează numărul de tacturi CPU utilizate de program până aici*/
- time_t time(time_t *tp); /*returnează nr. de secunde trecute de la 1 Inuarie 1970 */
- char *asctime(const struct tm *tp); /*convertește timpul inregistrat și pointat de tp într-un string ce reprezintă data*/
- char *ctime (const time_t *t_ptr); ?8 convertește timpul pointat de t_ptr la string - data*/
- struct tm *localtime(const time_t *t_ptr);

De exemplu:

```
…
time_t acum;
acum = time(NULL);
printf("%s\n%s ", ctime(&acum),
                asctime(localtime(acum)));
..
```

are ca efect scrierea datei calenderistice.

Codul ASCII

American Standard Code for Information Interchange										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	“	#	$	%	&	’
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Observații:

- Caracterul G este în linia 7 și coloana 1 deci are codul ASCII 71;
- Caracterele cu codurile 0-31 și 127 nu sunt tipăribile;
- Caracterul cu codul 32 este spațiu (gol) și se tipărește ca și un spațiu gol;
- Codurile caracterelor 0-9, A - Z și a - z sunt contigue;
- Diferența între codurile literelor mari și cele mici corespunzătoare este 32;
  În tabela următoare se da "semantica" unora din caracterele de pe liniile 0-2.

Înțelesul unor abrevieri			
bel	audible bell	ht	tab orizontal
bs	backspase	nl	linie nouă
cr	retur car	nul	null
esc	escape	vt	tab vertical

În sistemele UNIX, comanda **man ascii** produce afișarea tabelei codurilor ASCII pe ecran.