ANEXA B

NOȚIUNI DE C NECESARE DESFĂŞURĂRII LUCRĂRILOR DE LABORATOR

1. PREZENTARE TEORETICĂ

1.1 CONVENȚII

1.1.1 CONVENȚII DE SINTAXĂ

Deşi în momentul de față există un standard al limbajului C elaborat de ANSI (*American National Standard Institute*), nu toate compilatoarele îl respectă în totalitate. Unele dintre acestea (cum este *Borland*) oferă posibilitatea de a lucra şi în variante anterioare (ca de exemplu 'standardul' Kernighan-Ritchie, ce-şi ia numele de la cei care au creat limbajul prima dată, în laboratoarele Bell Labs. - AT&T în perioada 1965-1975).

Ceea ce este prezentat în această lucrare respectă standardul ANSI şi este acceptat de compilatoarele Borland, Microsoft precum şi a celor de tip open-source. Totuşi cele expuse pot fi folosite ca atare (ca o bază de studiu) şi pentru alte tipuri de compilatoare, deşi, pentru portabilitate, trebuie avute în vedere anumite rețineri (ce țin de implementarea şi specificul diferitelor variante). Consultați help-ul on-line sau documentația scrisă ce însoțeşte varianta de compilator pe care o folosiți !

Prezentarea generală a limbajul C şi a caracteristicilor sale cele mai importante se va face plecând de la *scheletul(structura)* unui program C, aşa cum este impus de către standardul ANSI (fig. 1).

A.[optional] - Eventuala descriere a scopului programului ce urmeaza (comentariu pe una sau pe mai multe linii) .

B.[optional/obligatoriu] - Zona fisierelor header(fisiere cu extensia '.h').

C.[optional] - Zona definitiilor de constante (#define), a declararii functiilor macro, a declararii sau definirii variabilelor globale si a definitiilor de tip (typedef).

D.[optional] - Zona prototipurilor de functii.

E.[optional/obligatoriu] - Functia 'main()'.

F.[optional] - Implementarea functiilor (corpul acestora) al caror prototip a fost anuntat in zona D.

Fig. 1 – Structura unui program ANSI C.

1.2 FIŞIERELE CE COMPUN UN PROGRAM C. DIRECTIVA '#include'

Orice program C constă dintr-o succesiune de *linii de program*, pe care utilizatorul le introduce cu ajutorul unui editor de texte. Mediul integrat C (***I**ntegrated **D**evelopment **E**nvironment*) pe care îl vom folosi la laborator conţine un astfel de editor. Pe măsură ce este introdus, textul programului este salvat într-un fişier numit *fişier sursă* având extensia: ***.c***, ***.cpp*** sau orice altă extensie se doreşte. Însă, preferabil este ca aceste fişiere să aibă extensia ***.c*** sau ***.cpp***, pentru a le diferenţia uşor de alte tipuri de fişiere.

Un program, funcţie de complexitatea sa, poate fi format din unul sau mai multe fişiere (numite şi *module*). Fişierele ce compun un program C pot fi: fişiere sursă şi *fişiere header*.

Scop: Fişierele header sunt acele fişiere ce conţin toate declaraţiile şi definiţiile de variabile, precum şi toate prototipurile de funcţii care îndeplinesc o sarcină comună. Extensia lor este '.h'(*h* de la header). De exemplu un fişier header poate conţine prototipurile funcţiilor ce execută operaţii legate de ecran sau cu tastatură, sau ale funcţiilor matematice.

Tipuri: Un fişier header poate fi:
- ***predefinit***, adică vine odată cu kit-ul compilatorului de C;
- ***definit de user***, adică creat de noi atunci când avem nevoie de un asemenea fişier.

Sintaxă: Despre fişierele header se spune că *se includ* în corpul programului pe care îl compunem. Astfel avem definite o serie de constante sau funcţii pe care le putem folosi, compilatorul ştiind astfel care funcţie trebuie apelată şi cu ce parametri, sau unde este corect să apară anumite variabile.

Includerea unui fişier header într-un program se face astfel:

```
#include <numeFisier.h>
```

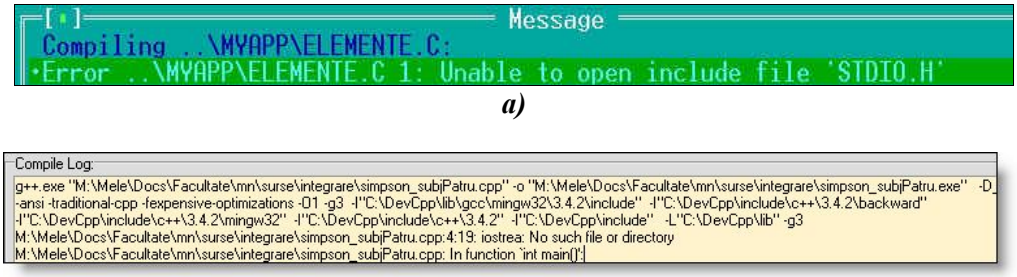
sau:

```
#include "numeFisier.h"
```

Obs.:
Cele două moduri nu ţin unul locul celuilalt (nu sunt echivalente !)

În ambele variante apare '`#include`'. Aceasta este o *directivă preprocesor*.

În *prima variantă*, pentru a face disponibile programului nostru definiţiile de variabile şi prototipurile de funcţii ale header-ului, compilatorul caută header-ul într-un director implicit de pe harddisk (funcţie de compilator), de exemplu `C:\BC\INCLUDE` pentru compilatoarele din familia Borland. Dacă nu este găsit se generează o **eroare de compilare**:

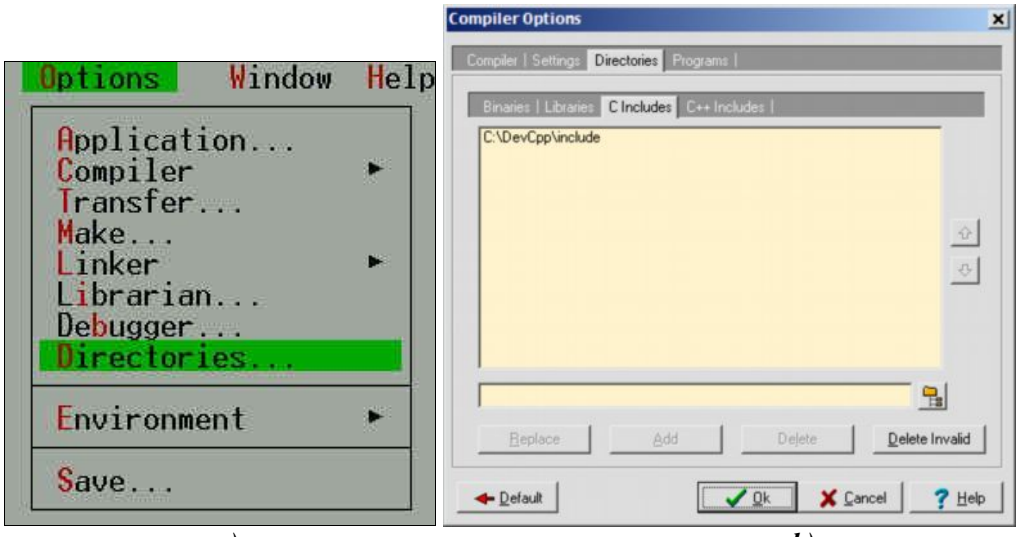


a)

b)

Fig. 2 – Eroare de compilare în cazul unui fişier header absent: a) cazul compilatoarelor din familia Borland C++; b) cazul compilatorului DeveloperC++

Directorul implicit se poate stabili prin intermediul opțiunii 'Directories' din meniul 'Options' (fig. 3-a, pentru compilatoarele Borland C++) sau din meniul Tools -> Compiler Options (fig. 3-b, pentru compilatorul DeveloperC++.



a) ***b)***

Fig. 3 – Stabilirea directoarelor de lucru ale compilatorului: a) familia BorlandC; b) DeveloperC++

A doua variantă caută mai întâi fişierul în directorul curent de lucru (acolo unde este salvat şi fişierul la care lucrăm). Dacă nu este găsit, abia atunci compilatorul îl caută în directorul implicit. Această variantă este mai flexibilă decât cea anterioară.

Orice fişier header, să spunem '`myHeader.h`', are asociat un fişier sursă '`myHeader.c`', în care sunt *definite* funcțiile ale căror prototipuri le avem în header (a defini o funcție înseamnă a-i stabili, prin corpul său de instrucțiuni, acțiunea efectivă

pe care o are de executat). Acest fişier sursă include **obligatoriu** fişierul header asociat. La rândul lui, un fişier header poate include şi alte fişiere header.

Observaţie:
Nu confundaţi perechea *fişier antet – fişier .c* asociat cu varianta de scriere obişnuită, în care programul la care lucrăm trebuie să apeleze anumite fişiere header, dar numele său nu este neapărat identic cu al fişierelor header pe care le apelează.

1.3 COMENTARII

Comentariile ***pot apare oriunde*** în interiorul unui program C. Ca urmare, aşa cum vedeţi în figura 1, un cod sursă C poate sau nu începe cu un **comentariu**.

Scop: Comentariile au în principal rolul de a lămuri programatorul (şi în general pe oricine citeşte un cod sursă) asupra a ceea ce s-a dorit a se executa într-un anume punct al programului, sau ce rol au anumite variabile sau instrucţiuni ale unui program.

Sintaxă: În C\C++ se acceptă comentarii de două tipuri:

- comentariu de linie (pe o linie):

```
// Un comentariu pe o singură linie .
```

Deci un comentariu pe o singură linie este precedat de două slash-uri (//).

- comentariu pe mai multe linii:

```
/* Un comentariu pe mai multe linii, ce începe aici …
 * … şi …
 * … se încheie pe linia imediat următoare.
 */
```

După cum se vede, un comentariu pe mai multe linii începe cu combinaţia '*slash-asterisc*' şi se încheie cu combinaţia '*asterisc-slash*'. Aceste două combinaţii sunt obligatorii. Compilatorul de C odată ce întâlneşte combinaţia `/*` consideră că tot ceea ce urmează este comentariu, până ce apare combinaţia `*/` . Între aceste două delimitatoare pot apărea ***orice*** caractere. În exemplul dat, asteriscurile intermediare pot lipsi. Ele nu au decât rolul de a arăta că un comentariu pe mai multe linii încă nu s-a terminat. Alinierea textului pe care am făcut-o în exemplu vrea să scoată în evidenţă tocmai acest rol.

Obs.: Un program bine scris ***trebuie*** să fie şi bine comentat !

1.4 DIRECTIVA '#define'

1.4.1 DEFINIȚIA CONSTANTELOR GLOBALE

Pentru a defini o constantă globală este nevoie de o directivă preprocesor:

```
#define
```

Anumite programe au nevoie de constante particulare, a căror existență este impusă de specificul aplicației. De exemplu aplicațiile de nivel scăzut (driver-e) pot avea nevoie de anumite constante pentru porturile (identificate prin adrese) specifice diverselor componente hardware.

Scop: Pentru a evita folosirea repetată în program a unei valori se optează pentru definirea ei o singură dată sub un nume sugestiv, şi folosirea în interiorul programului a acelei denumiri.

Sintaxă: Directiva '`#define`' are un loc unic unde poate să apară într-un program: *imediat după directivele* #include.

În general sintaxa acestei construcții este:

```
#define  numeConstanta  valoare
```

pentru constante numerice, constante caracter (încadrate între ghilimele simple) sau constante şir (încadrate între ghilimele duble).

Exemple:

a. Să definim două constante utile pentru calcule logice - TRUE şi FALSE:

```
#define TRUE 1
#define FALSE !TRUE     // semnul exclamării
                        // semnifică operatorul logic al
                        // negației(NOT).
…
if(var == TRUE) instructiune;
…
```

b. Definiția unei constante caracter şi a unei contante şir :

```
#define LITERA 'm'      // constanta caracter.
#define SIR "Nume"      // constanta sir.
```

Obs.:

1) Oriunde apare în program numele unei constante, compilatorul o substituie cu valoarea sa, conform definiției.
2) Se obişnuieşte ca numele constantelor să fie date cu majuscule pentru a face diferența față de celelalte nume ale variabilelor din program.
3) După întreaga construcție ***nu apare punct şi virgulă !***

1.4.2 FUNCŢII MACRO

O altă aplicaţie a directivei `#define` este în definirea *funcţiilor macro.*

Sintaxă:
Iată un exemplu de funcţie macro:
```
#define SIGN(a)  (a)>=0 ? 1 : -1
```

Operatorul '`? :`' este echivalentul blocului de deciziei '*if - else*'. Pentru exemplul ales construcţia se traduce astfel :
```
if (a>=0)  rez =1;
else rez = 0;
```

În program, acolo unde se întâlneşte numele funcţiei macro, acesta este înlocuit textual cu corpul din definiţie, unde pe post de 'a' este parametrul efectiv cu care a fost apelat macro-ul. Această înlocuire poartă numele de ***macroexpandare***. Rezultă că dacă într-un program se fac apeluri numeroase ale funcţiilor macro, codul sursă se măreşte, ducând la o încărcare a procesului de compilare.

Faţă de o funcţie obişnuită, un macro este avantajos în cazul programelor mici, unde *asigură rapiditate*. În rest, funcţia îşi păstrează avantajele nete:

- poate returna valori;
- poate asigura portabilitatea;
- evaluarea parametrilor efectivi se face o singură dată, la transferul lor către funcţie;
- se fac *verificări de tip* sau sintaxă asupra parametrilor transmişi, pe baza prototipului funcţiei.
- pot fi transmise prin referinţă orice tipuri de date definite de C sau definite de user.

Obs.:

1. Ca şi definiţia constantelor folosind directiva `#define`, şi definiţia unei funcţii macro nu se încheie cu punct şi virgulă ! În schimb, dacă o funcţie macro conţine mai multe instrucţiuni, atunci după fiecare se pune punctul şi virgula pe care compilatorul C le cere obligatoriu.
2. Dacă nu am fi inclus între paranteze numele variabilei pseudo-parametru 'a' am fi fost predispuşi la erori !

Exemplu:
Presupunem următoarea funcţie macro:
```
#define RAPORT(a,b)  a/b
```
împreună cu apelul său:
```
RAPORT(7+var << 2, 2);
```
Macroexpandarea ar duce la:
```
7+var << 2/2
```
Conform priorităţii operaţiilor, ceea ce se execută mai întâi este adunarea lui 7 cu '`var`'. Apoi ceea ce se obţine este deplasat la stânga cu 2 poziţii ('`operand 1<<`

`operand 2`' înseamnă deplasarea la stânga a operandului 1 cu operand 2 biți). În final se face împărțirea la 2. Dar nu aceasta am dorit !

Concluzia este:

Întotdeauna la o funcție macro, în corpul său trebuie ca argumentele să apară incluse între paranteze rotunde (operatorii cu precedența cea mai ridicată). Se asigură astfel că ordinea operațiilor este cea dorită de noi.

Vorbind în general, trebuie să ne asigurăm, cu ajutorul parantezelor rotunde, că ordinea operațiilor este cea dorită.

1.5 VARIABILE GLOBALE: DECLARARE ŞI DEFINIRE

Următoarea zonă a unui program C este zona în care se declară/definesc variabilele globale. Acestea se numesc **globale** deoarece apar în afara oricărui corp de funcție, fiind astfel ***cunoscute întregului program***.

Rol: Pot fi folosite fără *efecte laterale* în special în programele de dimensiune redusă, dar şi aici cu mare atenție !

Pe post de variabile globale pot fi variabile de orice tip C sau definit de utilizator.

Efectele laterale produse de astfel de variabile rezultă din însăşi faptul de a fi declarate global: ***oricine are acces la aceste variabile***, de unde rezultă că se poate întâmpla ca anumite modificări produse asupra variabilei de către funcția '`f1()`' din program să nu fie cunoscute de celelalte funcții ('`f2()`', '`f3()`' ş.a.m.d.).

Consecința este oarecum ascunsă: următoarea funcție care va folosi variabila (de ex. '`f2()`'), găseşte aici valoarea lăsată de '`f1()`'. Este foarte posibil ca această valoare să nu aparțină domeniului de intrare al funcției '`f2()`', de unde rezultă o comportare neaşteptată a acesteia, chiar dacă ea a fost corect concepută !

Sintaxă: Declararea/definirea variabilelor globale se face ca şi declararea/definirea oricăror altor variabile C, anume:

```
tipVariabilă numeVariabilă [= valoare];
```

În linia precedentă pe post de '`tipVariabilă`' poate fi:

- *un tip predefinit din C* (`char, int, float, double, long double`);
- *un tip compus*, care este creat pe baza cuvintelor-cheie şi a caracterelor semnificative din C.

Obs.: Nu uitați de punctul şi virgula de la sfârşitul declarației! În C orice declarație sau instrucțiune se încheie cu punct şi virgulă.

Exemplu:

```
int semnal; // declararea şi în acelaşi timp
      // definirea unei variabile de tip predefinit C
      // (aici 'int').
char *pCaracter;  // declararea unui pointer
      // către un caracter. Acesta este un tip compus
      // din cuvântul-cheie 'char' şi caracterul '*'.
```

O declaraţie în care se face şi rezervarea de spaţiu de memorie este numită ***definiţie***.

O declaraţie în care variabilei i se atribuie o anumită valoare la momentul declarării se numeşte ***iniţializare***.

În linia ce arată modul de declarare/definire, prezenţa parantezelor drepte semnifică faptul că ceea ce apare între ele este *opţional*. Este o convenţie de notaţie pe care o vom folosi şi mai departe.

Ex.: Varianta:

```
int semnal = 1;
```

reprezintă o *definiţie* şi în acelaşi timp o iniţializare. Definiţie, deoarece compilatorul rezervă spaţiu pentru a 'ţine' valoarea 1. Iniţializare deoarece se atribuie o variabilei o valoare în momentul declarării.

Varianta:

```
char vector[10];
```

este o *definiţie*, deoarece se alocă automat spaţiul necesar celor 10 componente ale vectorului.

1.6 TIPURI DE DATE DEFINITE DE UTILIZATOR

Pe lângă tipurile standard C, pe care orice compilator le pune la dispoziţia utilizatorilor, conform standardului ANSI, un utilizator îşi poate defini propriile tipuri, prin intermediul unor *definiţii de tip*. Această tehnică este conformă ANSI şi trebuie să fie implementată în orice versiune de compilator de C.

Rol: Definirea unor tipuri de diferite complexităţi, pe baza unei combinaţii de tipuri C de bază şi definite de user.

Sintaxă:

```
typedef  tipC tipUser;
```

În această sintaxă `tipC` reprezintă unul din tipurile C de bază, sau un tip definit anterior de user (tot cu ajutorul lui '`typedef`'). `tipUser` se spune că este un ***alias*** (o altă denumire) pentru tipul C respectiv. ***Această construcţie se încheie obligatoriu cu punct şi virgulă.***

Exemple:

1. În acest exemplu se dă o altă denumire tipului 'int' cunoscut din C **:**

```
typedef int INTREG; // INTREG este o altă
                    // denumire pentru 'int'.
…
INTREG a1, a2;     // declar două variabile întregi.
```

2. Un alt nume pentru un pointer la un tip predefinit C:

```
typedef float *pREALS;  // definirea unui
            //pointer la un tip real în simplă
            //precizie.
…
pREALS p1=&var1, p2=&var2;    // declararea a doi
      // pointeri la tipul real, inițializați
      // respectiv cu adresa variabilei 'var1' şi
      // a variabilei 'var2'.
```

3. Definirea unui nou tip de dată pe baza unui tip anterior definit tot de user:

```
typedef  pREALS *ppREALS; // definirea unui
      //pointer la un pointer la tipul real în
      //simplă precizie (adică un pointer dublu).
…
ppREALS p3=&p2; // declararea unui pointer la un
      // pointer la tipul real în simplă precizie,
      // inițializat cu adresa pointerului p2
      // anterior definit.
*p3 = &var3; // modific valoarea pointerului
      // dublu definit anterior.
**p3 = 2.1; // stabilesc, cu autorul
      // pointerului dublu, valoarea variabilei
      // 'var3'. Variabila 'var3' trebuie
      // declarată în prealabil !!
```

Obs.: Toate denumirile date de user este bine să fie notate cu litere mari, pentru a face mai uşor diferența între acestea şi alte tipuri sau variabile din program, ce apar scrise cu litere mici.

1.7 PROTOTIPURI DE FUNCȚII / CORPUL FUNCȚIILOR

1.7.1 PROTOTIP DE FUNCȚIE

Standardul ANSI impune ca orice variabilă sau funcție înainte de a fi folosită să fi fost:

- pentru variabilă: declarată/definită;
- pentru o funcție: să fi fost declarată, adică să-i fie cunoscut ***prototipul*** (care arată domeniul şi codomeniul acelei funcții).

Rol:

1. Crearea funcțiilor (de bibliotecă sau proprii aplicațiilor scrise de user) cu ajutorul fişierelor header. Rezultă de aici un rol implicit în modularizarea programelor, de unde flexibilitatea scrierii acestora.
2. Un prototip(adică ***declararea unei funcții***) este necesar şi atunci când user-ul nu găseşte un echivalent (o variantă) printre funcțiile de bibliotecă definite de C şi este nevoit să-şi creeze propria funcție.

Sintaxă:

```
tipReturnat  numeFuncţie(tip1 [arg1][, tip2[arg2]]);
```

'`tipReturnat`' este un tip implicit(standard) C sau definit de user. Numele funcţiei nu trebuie să conţină spaţii şi, ***la fel cu numele oricărei variabile***, poate începe doar cu o literă sau cu caracterul de subliniere - 'underscore'(_).

Notaţia cu paranteze drepte semnifică faptul că ceea ce apare între ele este opţional. Astfel, numărul argumentelor funcţiei poate fi *variabil.* Adică pot avea unul, două sau mai multe argumente. De asemenea, numele argumentelor pot lipsi.

În cazul prototipului – care semnifică ***declararea funcţiei*** – ne interesează ***doar tipul*** argumentelor de la apel şi nu numele lor, precum şi tipul întors de funcţie. Acestea sunt necesare în vederea *verificărilor de tip* pe care le face compilatorul în momentul:

- ***transferului parametrilor***: acesta este procesul prin care *parametrii formali* sunt înlocuiţi – folosind stiva – cu *valorile argumentelor efective* de la apel.
- ***returnării valorilor***: acesta este procesul prin care funcţia îşi încheie execuţia cu specificarea unei valori returnate. Această valoare trebuie să fie *de acelaşi tip* cu cel din declaraţia funcţiei, în caz contrar semnalându-se eroare.

În prototip se pot specifica şi numele parametrilor formali, fără a constitui o greşeală. Compilatorul având aici nevoie doar de tipul argumentelor, va ignora numele pe care le veţi da.

Exemple:

1. *Declararea* unei funcţii care preia, în această ordine, un întreg şi adresa unui '`char`', şi întoarce o valoare de tip întreg:
   ```
   int lungimeSir(int, char *);
   ```

2. Varianta anterioară modificată în care se specifică şi numele parametrilor formali:
   ```
   int  lungimeSir(int dim, char vector[]);
   ```

Atenţie ! Un vector este complet definit prin *tripletul*:
`{dimensiune, adresaPrimElement, tipElementeVector}`.

De aceea nu este necesar să se specifice explicit dimensiunea vectorului între parantezele drepte. Dacă primul argument lipseşte atunci compilatorul semnalează eroare, deoarece nu se specifică dimensiunea vectorului.

3. Prototipul unei funcţii care preia o matrice de elemente reale:
   ```
   float maxMatrice(float matrice[DIML][DIMC]);
   ```

este echivalent cu:

```
float maxMatrice(int, int, float **);
```

1.7.2 CORPUL FUNCȚIILOR

Punctele D. şi F. din structura unui program ***C*** sunt condiționate în modul următor:

dacă D atunci F

Ceea ce înseamnă că dacă nu am un prototip atunci, evident, nu are rost să definesc un corp de funcție.

Rol: Corpul funcției este echivalentul ***definiției funcției***.

Sintaxă: Atenție! Se aseamănă foarte mult cu prototipul funcției. Diferențele sunt următoarele:

- punctul şi virgula este înlocuită cu combinația care în C specifică ***un bloc de instrucțiuni***:

```
{
  [declarații/definiții;]
  instrucțiune;
  [instrucțiune;]
}
```

- este obligatorie specificarea numelor parametrilor formali, deoarece urmează să-i folosesc în interiorul funcției. Conform standardului ANSI-C, orice parametru al unei funcții este o variabilă locală funcției care îl declară.

Iată *sintaxa generală*:

```
tipReturnat numeFuncție(tip1 numeArg1[,tip2 numeArg2])
{
   [declarații/definiții;]
   instrucțiune;
   [instrucțiune;]         // se poate repeta sau nu.

  [return expresie;] // poate lipsi.
}
```

O funcție poate să nu aibă nevoie de alte variabile locale (declarate în interiorul corpului funcției). În acest caz declarațiile/ definițiile de variabile lipsesc.

Dacă o funcție nu returnează nici o valoare (adică '`tipReturnat`' este '`void`'), atunci instrucțiunea '`return expresie;`' ***trebuie*** să lipsească (nu are un corespondent în tipul valorii returnate).

Exemple:

1. Funcție care nu returnează nici o valoare:

```
void  mesaj(char *sir)
{
      puts(sir);
}
```

Aici se preia un şir care se tipăreşte. Funcția nu trebuie să întoarcă nici o valoare.

2. Funcţie care întoarce obligat o valoare:

```
int maximCaracter1(char ch1, char ch2)
{
if(ch1 > ch2) return 1; // test de cod ASCII
else  if(ch1 = = ch2) return 0;
      else return 2;
}
```

În urma comparării codului ASCII asociat fiecărui caracter se stabileşte dacă primul caracter este mai mare decât al doilea (cu returnarea valorii 1), dacă sunt egale(returnarea lui 0) sau al doilea este mai mare decât primul (returnarea valorii 2).

3. O altă variantă a funcţiei de mai sus este aceea în care se defineşte o variabilă locală numită '`test`' a cărei valoare o stabilim în funcţie de rezultatul logic al testelor efectuate. Apoi returnăm această valoare.
   Această variantă este:

```
int maximCaracter2(char ch1, char ch2)
{
      int test; // definirea unei variabile locale în
            // care ţin o valoare particulară determinată
            // de rezultatul testelor logice.
      if(ch1 > ch2)
      {     // test de cod ASCII
            test = 1;
            return test;
            }
      else  if(ch1 = = ch2)
            {
                  test = 0;
                  return test;
                  }
            else {
                  test = 2;
                  return test;
                  }
}
```

Între variantele de la 2. şi 3. observaţi că declararea/definirea variabilelor locale funcţiei poate lipsi.

1.8 FUNCŢIA 'main()'

Rol: Un program are capacitatea de a rula doar dacă conţine *funcţia principală*. **Aceasta este unică !**

Particularităţile acesteia sunt date în continuare.

Sintaxă:

Ca orice funcţie, şi funcţia '`main()`' poate returna o valoare şi poate prelua parametri. Doar că aici există ***restricţii foarte mari.***

- tipul returnat: `void, char, unsigned char, int, unsigned int`;
- tip preluat: `void` sau combinaţia {`int, char**`} .

Exemple:

1. Funcţie `main()` care nu întoarce nici o valoare şi preia argumente din linia de comandă:
   ```
   void main(int argc, char *argv[]) { … };
   ```

2. Funcţie `main()` care întoarce un întreg şi nu preia nici un parametru:
   ```
   int main(void) { … };
   ```

3. Funcţie `main()` care nu preia argumente şi nici nu întoarce vreo valoare:
   ```
   void main(void) { … };
   ```

2. NOTE ADIŢIONALE

2.1 OPERATORI

Limbajul C posedă o serie de operatori ierarhizaţi conform *priorităţii* lor. Aceştia împreună cu *operanzii* constituie *expresiile*.

Parantezele rotunde sunt operatorii cu prioritatea cea mai mare. Deci cu ajutorul lor putem specifica după cum dorim ordinea de execuţie a operaţiilor în cadrul unei expresii.

Tip: Operatorii pot fi clasificaţi:

A. *După numărul operanzilor*:

- **unari**: adică necesită un singur operand pentru a–şi îndeplini acţiunea;
- **binari**: necesită doi operanzi pentru a fi complet definiţi;
- **ternari**: acţionează asupra unei combinaţii de trei operatori.

Exemple:

Unari: +, - (schimbarea semnului), ++ (incrementare), -- (decrementare), operatorul de conversie explicită – (cast), ! (complement faţă de 2 – negaţia logică), *(dereferenţiere), &(luarea adresei), sizeof() (operatorul de determinare a dimensiunii în octeţi în memorie a operandului asupra căruia acţionează).

Binari: + (adunare), - (scădere), *(înmulţire), / (împărţire), % (restul împărţirii a doi întregi), << (deplasare stânga), >> (deplasare dreapta), == (testarea identităţii operanzilor), != (diferit), & (AND bit cu bit), | (OR bit cu bit), ^(XOR- SAU exclusiv bit cu bit), ~ (complement faţă de 1 – negare bit cu bit), && (AND

logic), ||(OR logic), > (mai mare), >= (mai mare sau egal), < (mai mic), <= (mai mic sau egal), = (atribuire – cu variantele: +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=), operatorul virgulă.

Ternar: ?: (test 'if-else' rapid): cond? rez1 : rez2
În instrucţiunea:

```
rezultat =(cond? rez1: rez2);
```

explicaţia sa este următoarea:

```
if(cond) rezultat = rez1;// dacă condiţia este
         //evaluată adevărat, atunci rezultatul este
         //'rez1' .
else rezultat = rez2;
```

B. *După tipul rezultatului:*
- operatori aritmetici;
- operatori relaţionali şi logici;
- operatori pe bit;
- alţi operatori.

Prima categorie are operanzi de tip întreg, real şi pointer. A doua categorie conţine operatorii de comparaţie (mai mic, mai mare şi variantele cu egal) şi operatorii logici(SAU, ŞI, SAU EXCLUSIV, NOT). Operatorii pot fi de tip caracter, întreg, real şi de tip adresă. Cea de-a treia categorie conţine variantele de operatori logici pe bit. Operatorii sunt consideraţi în reprezentare binară(sau octală, hexazecimală). În cea de-a treia categorie intră operatorii de luare a adresei (referinţă), de dereferenţiere, parantezele rotunde (care au semnificaţia de apel de funcţie), parantezele drepte (cu rolul de indexare a tablourilor) , operatorul virgulă (care are efect de secvenţiere), operatorul de conversie de tip explicită, operatorul de determinare a numărului de octeţi al reprezentării în memorie a operanzilor.

Exemple:

```
1. f(x);        // apelul unei funcţii
2. v[3] = 'd';       // indexarea tablourilor.
3. spatiu = sizeof(int);   // în variabila 'spaţiu' am
                // numărul de octeţi ai tipului întreg cu
                // semn din C.
4. final = (float) varIntreg;    // conversie explicită de
          // la tipul întreg al variabilei 'varIntreg' la
          // tipul real în simplă precizie.
          // variabila 'final' trebuie să fie de tipul către
          // care fac conversia.
```

2.2 DECLARAREA ŞI DEFINIREA POINTERILOR

Pointerul înseamnă ***adresă***. Pointerul este o variabilă în care se ţine adresa unei alte variabile de un anumit tip. Ca orice alta variabilă, şi o variabilă pointer *are asociată o adresă*.

Rol:

În C vectorii pot fi luaţi ca parametru al funcţiilor doar prin pointeri, adică prin intermediul adresei primului element al vectorului. Compilatorul este astfel construit încât orice apel al unei funcţii cu parametru vector este tradus la un apel prin referinţă. Un vector nu poate fi returnat de o funcţie. Ceea ce poate face însă o funcţie este să returneze pointerul la primul element al vectorului.

Funcţiile pot lua alte funcţii ca parametru *doar prin pointeri la funcţii*. O funcţie poate întoarce – relativ la funcţii – doar un pointer la o funcţie de un anumit tip.

Alocarea dinamică a memoriei se face inevitabil cu ajutorul pointerilor.

Apelul prin referinţă este singura modalitate prin care modificările asupra variabilelor ce au loc în interiorul funcţiilor să se reflecte şi după ce acestea se încheie.

C permite existenţa pointerilor către orice tip (standard C sau definit de user).

Sintaxă:

```
tip *numePointer [=adresaInit];
```

Asteriscul poate fi legat fie de `tip` fie de `numePointer`.

Este bine să fiţi consecvenţi în notaţie şi să alegeţi una dintre cele două variante pe care să o folosiţi de fiecare dată.

'`tip`' poate fi standard C sau definit de user.

Un pointer poate să fie sau nu iniţializat. Totuşi trebuie ca *întotdeauna* un pointer să puncteze undeva în memorie. Altfel au loc erori aleatoare de program, care sunt dificil de depistat.

Iniţializarea unui pointer se face cu ajutorul operatorului de ***luare a adresei*** (&). Acesta trebuie urmat de numele variabilei cu a cărei adresă iniţializăm pointerul. Opusul acestui operator este cel *de dereferenţiere* (*). Prin intermediul acestui operator lucrăm asupra ***valorii variabilei*** (***a conţinutului***) a cărei adresă o ţine pointerul(conţinutul zonei de memorie asociată variabilei).

2.2.1 VALOAREA NULL

Există şi situaţia în care un pointer este de tip `void`, adică stabilit *să nu puncteze nicăieri*, valoarea sa fiind `NULL` (aceasta este o constantă predefinită în limbajul C, cu valoarea 0). Dacă de exemplu în urma unei alocări dinamice de memorie valoarea returnată este NULL aceasta înseamnă *eroare de alocare*. Compilatorul marchează eroare setând valoarea pointerului la NULL. La fel, user-ul poate folosi această valoare în mod explicit şi conştient.

Eroarea pe care o poate face user-ul ***în mod inconştient*** este aceea de a nu iniţializa pointerii cu care lucrează. Un pointer neiniţializat punctează în memorie mereu în alt loc (deci aleator), la fiecare rulare a programului. De aici provine dezastrul: este posibil ca zona la care punctează să aparţină sistemului de operare.

Programul folosind acea zonă influenţează negativ sistemul de operare şi acesta se va comporta anormal (blocări, reset-ări etc.).

2.2.2 ARITMETICA POINTERILOR

A. Un pointer poate fi indexat aşa cum numele vectorului primeşte între paranteze drepte o valoare constantă numită ***index***. Acest proces este numit ***indexare***. Prin indexare ***se adună*** de fapt la adresa pe care o ţine pointerul un număr egal cu:

```
index*sizeof(tip)
```

adică un multiplu întreg de numărul de octeţi ai tipului către care a fost declarat pointerul.

B. Singura operaţie care mai este acceptată asupra pointerilor este ***scăderea***. Evident, se pot scădea doar pointeri către acelaşi tip. Semnificaţia este aceea a numărului de octeţi dintre două adrese de memorie (care pot fi sau nu alăturate). Dacă această operaţie este definită de implementarea compilatorului cu care se lucrează, atunci are rost şi compararea pointerilor, deoarece la baza acestei operaţii stă scăderea. Astfel:

```
p1 < p2  se traduce în:  p1-p2<0 .
```

Exemple:

1. Declararea unui pointer (linia ce urmează constituie o eroare intrinsecă *în cazul* folosirii pointerului `pFloat`, din cauza neiniţializării acestuia):

```
float *pFloat; // 'pFloat' punctează la o adresă
         //aleatoare. Conţinutul va fi şi el aleator.
```

2. Definirea unui pointer (se primeşte valoare de iniţializare):

```
int var;  // se defineşte variabila întreagă
          // 'var'. În acest moment ea posedă o adresă,
          // alocată de către compilator.
int *pInt = &var;     // definirea pointerului:
          // acesta ia ca valoare iniţială adresa
          // variablei de tip întreg 'var'.
```

3. Lucrul asupra conţinutului zonei de memorie asociate variabilei punctate de pointer:

```
// definirea unui vector de două caractere.
char vector[2] = {'a', 'b'};
char *pChar = vector;      // pointerul 'pChar'
   // reţine adresa primului element al vectorului.
*pChar = 'c'; // echivalent cu: vector[0] = 'c';
```

4. Testul asupra erorii de alocare dinamică a unei zone de memorie:

```
int pInt = (int *)malloc(2*sizeof(int));
if(pInt == NULL)
{
      printf("\n Eroare de alocare a memoriei !");
      printf("\n Reluați programul.");
}
```

5. Indexarea unui pointer: (presupun definit vectorul de la ex. 3 de mai sus)

```
float *pChar = vector;
*(pChar + 1) = 'd';
            // echivalent cu: vector[1] = 'd';
            // Cum pointerul arată către primul element
            // al unui vector de caractere, adunarea cu
            // 1 semnifică adăugarea la valoarea
            // pointerului a cantității 1*sizeof(char)
            // (adică 1 octet). Se punctează, deci, al
            // doilea element al vectorului.
```

2.3 OPERAȚII ASUPRA FUNCȚIILOR

O funcție, ca şi o variabilă, poate fi ***declarată*** şi ***definită***, aşa cum am arătat. Orice funcție posedă o adresă în memorie, adresă ce punctează către începutul funcției.

2.3.1 APELUL FUNCȚIILOR

O funcție odată definită trebuie să aibă un rol în cadrul programului ce o foloseşte. Activarea unei funcții se numeşte *apel de funcție*. Este normal ca, înainte de apel, o funcție să aibă pregătiți toți parametrii, astfel încât, atunci când începe să lucreze, aceasta să folosească valorile *parametrilor efectivi* pe care îi primeşte.

Obs.: Faceți diferența între *parametrii formali* şi *parametrii efectivi* ai unei funcții. Parametrii formali sunt cei anunțați în declarația funcției. Numele lor nu contează, putând fi ales după dorință. Aceştia nu pot fi folosiți ca atare. Parametrii efectivi sunt acei parametri ale căror valori sunt preluate de către parametrii formali ai funcției, în momentul apelului.

La apel, între parametrii efectivi şi cei formali trebuie să existe coincidență:

- **în tip**: trebuie ca oricare parametru efectiv să fie de tip identic sau compatibil cu tipul corespunzător parametrului formal din declarație pe care urmează să-l inițializeze;
- **în număr**: numărul parametrilor efectivi trebuie să respecte numărul parametrilor formali din declarația funcției.

Dacă măcar una din condițiile de mai sus nu este respectată compilatorul semnalează eroare.

2.3.1.1 TRANSFERUL PARAMETRILOR

Procesul prin care valoarea parametrilor efectivi este preluată de funcție (prin copiere în parametrii formali, pentru care ***se alocă dinamic*** spațiu pe stivă) se numeşte *transfer de parametri*.

În C, *dacă nu se specifică altfel*, transferul parametrilor se face ***prin valoare***. Acesta este cazul transferului tuturor tipurilor standard C sau definite de user, *cu excepția tipurilor tablou şi funcție*. Pentru acestea din urmă, atât apelul cât şi returnarea unei valori către aceste tipuri se face ***prin referință***. Referință înseamnă că sunt folosiți pointerii. User-ul poate însă, în mod explicit, să transfere prin referință parametri de orice tip definit de el(cu ajutorul `typedef`) sau standard C.

Odată transferați, parametrii efectivi *inițializează* parametrii formali. În corpul funcțiilor sunt folosite numele parametrilor formali pe care i-am dat la momentul definirii funcției, dar valorile lor:

- pentru *transferul prin valoare* sunt ***cópii*** ale parametrilor efectivi. Consecința este aceea că, dacă se modifică vreunul din parametrii efectivi în corpul funcției, noua valoare nu se reflectă în parametrul efectiv respectiv, ci doar copia este alterată. La ieşirea din funcție, parametrul efectiv rămâne la valoarea din momentul apelului funcției. Acest mecanism se numeşte *apel prin valoare*, şi după cum este implementat compilatorul de C, o acțiune asupra unui parametru efectiv trimis prin valoare nu se reflectă în acel parametru după ce funcția se încheie;
- pentru transferul prin referință sunt ***adresele*** parametrilor efectivi. Orice modificare, intervenită în cadrul funcției, asupra conținutului zonei de memorie a cărei adresă a fost preluată ca parametru efectiv, se reflectă în valoarea acelui parametru şi după încheierea funcției. Acesta este apelul prin referință.

Exemple:

1. Funcție ce are declarați doi parametri formali, primul de tip `int` şi al doilea de tip `char*`. Funcția nu returnează nici o valoare:

```
void fct1(int, char *); // primul parametru formal
         // este de tip int , iar al doilea este un
         // pointer la un caracter.
…
fct1(dim, vectChar);    // apelul funcției cu doi
         // parametri efectivi, anterior definiți: un
         // întreg 'dim' şi adresa de început a
         // vectorului de caractere 'vectChar'. Tipul
         // lor trebuie să respecte tipul parametrilor
         // formali, respectiv.
```

2. O funcţie pentru care nu se respectă numărul parametrilor efectivi:

```
int fct2(char, float);
…
char ch = 'a';
int rez = fct2(ch);     // Eroare ! Funcţia a fost
      // declarată ca având doi parametri, iar la apel
      // folosesc doar unul.
```

3. Acelaşi exemplu ca mai sus, doar că nu am iniţializat caracterul `ch` înaintea apelului funcţiei. Rezultă o eroare cu efect secundar, deoarece valoarea caracterului va fi aleatoare de la o rulare la alta a programului.

```
int fct2(char, float);  // declaraţia funcţiei.
…
char ch;    // trebuie iniţializat înainte de apelul
         // funcţiei care il floseste ca parametru !
float număr = 1.3;
int rez = fct2(ch, numar); // apelul funcţiei.
      // 'ch' si 'numar' sunt parametrii efectivi, ce
         // îsi transfera valoarea catre funcţie,
         // respectând tipul şi numărul parametrilor
         // formali. Aceşti parametri efectivi trebuie sa
         // conţină valori în momentul apelului funcţiei !
```

4. Funcţie cu apel prin referinţă. Pentru aceasta sunt folosiţi pointerii:

```
int nrElemVector(char v[]);  // declaraţia.
…
int nrElem;
// definiţia unui vector de 3 caractere
char vChar[]={ '1', '2', '3'}; .
nrElem = nrElemVector(vChar); // apelul funcţiei,
      // folosind transferul prin referinţa. Aici are
      // loc un transfer prin referinţa, prin care se
      // preia adresa de inceput a vectorului.
```

5. Funcţie ce preia un pointer la o funcţie de tip întreg şi cu argumente întregi, şi întoarce un pointer la o funcţie de tip `void` şi cu argument de tip şir de caractere (`char *`):

```
// declaraţia funcţiei a cărei adresă o returnează
// funcţia de bază.
void myFunctionReturn(char *);

// declaratia funcţiei a carei adresa este preluata de
// catre functia de baza.
int myFunctionApel(int, int);

// declaratia functiei de baza.
void (* fct3(int (*pFct)(int, int)))(char *);
…
char *mesaj = "Un mesaj.";
int (*pFctParam)(int, int) = myFuncţionApel;
```

```
            // un pointer la o functie initializat.
void (*pFctReturn)(char *); // declaratia unui
          // pointer la o functie ce preia un sir de
          // caractere si nu returneaza nimic.
pFctReturn = fct3(pFctParam); // apelul funcţiei de
        // baza. Aceasta, în cadrul corpului său, declară
        // o variabila pointer la o functie ce preia un
        // sir de caractere si nu returneaza nimic, şi
        // set-eaza aceasta variabila sa puncteze catre
        // funcţia 'myFunctionReturn()'. Apoi returneaza
        // aceasta variabila.
(*pFctReturn)(mesaj);   // apelul unei funcţii
         // prin intermediul unui pointer la funcţie.
```

6. Funcţie ce preia un întreg şi returnează un pointer la un `char`:

```
char *fct4(int); // declaraţia funcţiei.
…
int alegere  = 2;
char *msgReturnat;  // retin adresa mesajului returnată
      // de funcţie.
char *mesaje[] = {
      "Mesaj1",
      "Mesaj2",
      "Mesaj3"
      };    // definiţia unui vector de 3 siruri a
            // 6 caractere(de exemplu mesajele
            // posibile).
msgReturnat = fct4(alegere);// apelul funcţiei.
puts(msgReturnat); // afisarea mesajului selectat de
      // functie pe baza variabilei intregi 'alegere'.
```

Observaţii importante:

1. Parametrii formali sunt consideraţi ***variabile locale funcţiei***, adică necunoscuţi în afara corpului funcţiei. Nu mai este necesară deci, declararea lor în cadrul corpului funcţiei.

2. Transferul parametrilor se face cu ajutorul stivei, *în ordine de la dreapta spre stânga*. Astfel primul parametru formal al unei funcţii este situat în vârful stivei în momentul în care funcţia îşi începe execuţia. Acest mod de transfer al parametrilor se numeşte ***convenţia C de apelare*** al funcţiei. Mai există ***convenţia Pascal***, care este inversă – de la stânga la dreapta: primul parametru preluat pe stivă este chiar primul parametru al listei de parametri formali, astfel că la începutul execuţiei, în vârful stivei este situat ultimul parametru din lista parametrilor formali.

2.3.2 POINTERI CĂTRE FUNCȚII

Limbajul C permite programatorului *să generalizeze* codul scris, astfel încât sarcina scrierii funcțiilor de bibliotecă să fie mult mai uşoară.

În această tehnică este cuprins pointerul la funcție. Construcția ce urmează să fie prezentată este una specială. Prin aceasta, nu trebuie cunoscut efectiv din partea utilizatorului codul sursă al unei funcții de bibliotecă.

Să ne imaginăm următoarea *situație*:

Presupunem că se cere o funcție de bibliotecă, construită astfel încât să permită derivarea ***oricărei*** funcții reale de variabilă reală. În mod normal, numele funcției este diferit, în funcție de preferințele şi de posibilele convenții respectate de fiecare programator. Totuşi, atunci când codul urmează să devină comercial, aceste proprii obişnuințe trebuie în cea mai mare măsură adaptate politicii companiei care va lansa un anumit produs.

Aşadar, ce tehnică adoptă programatorul, astfel încât să 'ascundă' codul sursă, şi să nu permită unui utilizator sau altuia să schimbe instrucțiunile funcției pentru zonele unde numele funcției prevăzut inițial de către programatorii firmei nu coincide cu preferința utilizatorului acelei rutine.

Varianta adoptată poate fi cea în care esențial este rolul pointerului la funcție.

Etapele scrierii corecte a codului sursă sunt următoarele:

1. ***Declarația funcției către care urmează să puncteze pointerul***

Se pleacă de la semnătura funcției (denumită în C *prototip*, sau *declarație de funcție*). Astfel, dacă funcția care urmează să fie derivată este de tipul:

```
f:R -> R
```

deci funcție reală de variabilă reală, atunci în C acest lucru se scrie:

```
real numeFunctie(real); // prototip de funcție
```

unde real reprezintă unul din tipurile care modelează tipul real în C (`float, double, long double`).

2. ***Declarația pointerului***

Plecând de la semnătura funcției anunțată anterior, declarația pointerului devine naturală:

```
float (*pNumePointer)(float);
```

şi se observă că atât tipul argumentului cât şi tipul valorii returnate sunt întocmai cele impuse de către funcție.

Trebuie făcută aici **observația esențială** că nu se poate construi un pointer generic, de aşa natură încât să permită adresarea oricărei funcții posibile. Din acest motiv, succesiunea paşilor în construcția pointerului este cea prezentată.

3. *Inițializarea pointerului*

Conform teoriei pointerilor, odată construiți, există dedicați doi operatori specifici, pe care limbajul îi pune la dispoziție: operatorul referință (&) şi operatorul de indirectare (*).
Pentru inițializarea pointerului, sintaxa este următoarea:

```
pNumePointerFct = &numeFunctie;
```

sau, echivalent:

```
pNumePointerFct = numeFunctie;
```

4. *Apelul indirect de funcție, prin intermediul pointerului*

În sfârşit, odată construit, pointerul poate permite generalizarea cerută, permițând apelul funcției fără a mai folosi numele acesteia. Acest comportament este, dealtfel, cel consacrat pentru pointeri, vorbind în general. Se ştie că odată construit un pointer (către o variabilă de un anumit tip) modificarea valorilor acelei variabile sau determinarea valorilor acelei variabile se pot face fără a mai folosi numele acelei variabile, ci doar numele pointerului (folosind operatorii tipici pointerilor).

Şi pentru funcții există un comportament similar, astfel încât se poate apela o funcție fără a mai folosi numele cu care a fost declarată acea funcție.
Sintaxa este următoarea:

```
pNumePointer(x) <=> (*pNumePointer)(x) <=> numeFunctie(x)
```

Deci, oriunde apare numele funcției, acesta poate fi substituit cu numele pointerului, într-una din variantele prezentate. Aceste variante sunt perfect echivalente şi acceptate fără probleme de către compilator.

Urmând aceşti 4 paşi, se poate obține generalizarea funcțiilor de bibliotecă, asigurându-se astfel portabilitatea limbajului C.

2.3.3 ALTE REGULI

Pentru funcțiile care returnează anumite valori, acestea pot fi folosite în diferite contexte, determinate de regulile de lucru asupra tipurilor datelor returnate. Aceste contexte pot fi: expresii, condiții de cicluri, pe post de parametri ai apelurilor altor funcții.

De asemenea se poate ignora rezultatul întors de o anumită funcție. Un exemplu este funcția `printf()` care întoarce numărul de valori tipărite.

O funcție C ***nu poate conține definiția unei alte funcții***.

O funcție poate apela alte funcții anterior declarate, iar în particular o funcție se poate apela pe ea însăși, proces numit *recursivitate.*

Exemple:

1. Rezultatul unei funcții este folosit într-o expresie:

```
char choice;
choice = alegereUser(); // funcția 'alegereUser()'
         // întoarce un caracter dorit de utilizator.
         // Aceasta valoare returnata este tinuta în
         // variabila 'choice'.
…
```

2. Exemplul anterior modificat. Rezultatul întors de funcție este folosit dinamic în condiții – aici în cazul unui `if()`:

```
char choice;
if((choice = alegereUser())!= ESC) && (choice == 'a'))
      (*pFct1)(choice);
…
```

3. Expresie în care este folosit rezultatul unei funcții:

```
unsigned int par(int);
// funcția returnează un întreg fără semn:
// 0 dacă numărul este par, și 1 dacă el este impar.
…
sum = (f(ls)+f(ld))*h/3.;
for(i=1; i<=n; i++) sum += (2*h/3.)*(1+par(i))*valoare;

//  'valoare' se înmulțește:  cu 2h/3, pentru i par
//                            cu 4h/3, pentru i impar.
…
```

3. TIPURILE DE BAZĂ DIN C

Numele tipului	***Spaţiul de memorie (Bytes)***	***Gama de valori***
Char (implicit signed)	1	-2^7, $2^7 - 1$
Unsigned char	1	0, $2^7 - 1$
Int (implicit short şi signed)	2	-2^{15}, $2^{15}-1$
unsigned int (implicit short)	2	0, $2^{15}-1$
long int (implicit signed)	4	-2^{31}, $2^{31}-1$
unsigned long int	4	0, $2^{31}-1$
float (numai cu semn)	4	Real în simplă precizie.
double (numai cu semn)	8	Real în dublă precizie.
long double	10	Real în dublă precizie extinsă.
void	-	Are semnificaţia de nimic sau orice, funcţie de context.
tip *pTip (tipul pointer)	Spaţiul asociat lui tip	-
tip tablou[DIM] (tipul vector 1-dimensional)	DIM*sizeof(tip)	-
tip tablou[Dim1][Dim2]…[DimN] (tip tablou multidimensional)	Dim1*Dim2*Dim3*…*DimN*sizeof(tip)	-
struct nume{ tip1 câmp1; tip2 câmp2; …}	sizeof(struct nume) (se ţine cont de eventualele alinieri în memorie)	-
enum nume{ const1[=init1][, const2[=init2],…]} (tipul enumerare)	Compilatorul tratează valorile enumerare drept întregi.	-
union nume{ tip1 arg1[, tip2 arg2,…] }(tipul uniune)	La un moment dat se foloseşte doar una dintre variabilele pe care le cuprinde uniunea. Spaţiul de memorie alocat este dat de: Max(sizeof(tip1), sizeof(tip2),…)	-

4. LISTA SPECIFICATORILOR DE FORMAT ŞI A MODIFICATORILOR ACESTORA

4.1. SPECIFICATORII DE FORMAT

În figurile 4 şi 5 sunt prezentate combinaţiile specifice operaţiilor de intrare/ieşire. Operaţiile acestea au de-a face cu ceea ce se numeşte consolă, adică perechea tastatură-ecran.

Funcţiile de bibliotecă consacrate sunt cele din familia `printf()` şi `scanf()`.Sunt prezentaţi pe rând specificatorii fiecărei funcţii în parte.

În esenţă, specificatorii sunt construcţii ce îndrumă aceste funcţii în interpretarea datelor preluate. După cum se vede şi din liste, limbajul C oferă astfel de construcţii pentru fiecare tip de dată fundamental: familiile de numere **N**, **Z**, **R**, tipul pointer şi şirul de caractere. Pentru celelalte tipuri de date predefinite, din cauza faptului că ele pot fi compuse din cele amintite aici, nu s-au construit specificatori speciali.

Lista următoare prezintă specificatorii cunoscuţi de către `printf()`. Există şi funcţia `fprintf()`, concepută special pentru lucrul cu stream-uri, adică tipul abstract ce apare în lucrul cu fişierele.

%c	Caracter
%d	Numere întregi în baza 10, cu semn
%i	Numere întregi în baza 10, cu semn
%e	Notaţie ştiinţifică (cu litera e mică)
%E	Notaţie ştiinţifică (cu litera E mare)
%f	Număr zecimal în virgulă mobilă
%g	Foloseşte %e sau %f, care din ele este mai mic
%G	Foloseşte %E sau %f, care din ele este mai mic
%o	Număr în octal fără semn
%s	Şir de caractere
%u	Numere întregi zecimale fără semn
%x	Numere hexazecimale fără semn (cu litere mici)
%X	Numere hexazecimale fără semn (cu litere mari)
%p	Afişează un pointer
%n	Argumentul asociat este un pointer de tip întreg în care a fost plasat numărul de caractere scrise până atunci
%%	Afişează un semn %

Fig. 4 - Lista specificatorilor de format utilizaţi în operaţiile de intrare/ieşire de către funcţiile din clasa `printf()`.

Urmează, în figura 5, o listă similară, dar pentru funcţia `scanf()` şi cele din familie. Similar lui `fprintf()` există funcţia pereche `fscanf()`, dedicată lucrului cu fişiere: citirea datelor.

%c	Citeşte un singur caracter.
%d	Citeşte un număr întreg zecimal.
%i	Citeşte un număr întreg zecimal.
%e	Citeşte un număr în virgulă mobilă.
%f	Citeşte un număr în virgulă mobilă.
%g	Citeşte un număr în virgulă mobilă.
%o	Citeşte un număr în octal.
%s	Citeşte un şir.
%x	Citeşte un număr în hexazecimal.
%p	Citeşte un pointer.
%n	Primeşte o valoare egală cu numărul de caractere citite până atunci.
%u	Citeşte un număr întreg fără semn.
%[ ]	Caută un set de caractere.

Fig. 5 - Lista specificatorilor de format utilizaţi în operaţiile de intrare/ieşire de către funcţiile din clasa `scanf()`.

4.2. MODIFICATORII DE FORMAT

Pentru tipurile de date numerice, anunţate în paragraful 4.1 (**N**, **Z** şi **R**) există următorii modificatori de format:

- `h` - doar pentru tipurile întregi (`short`) ;
- `l` - long (atât pentru valorile întregi, cât şi pentru reale);
- `L` - doar pentru tipurile reale (`long double`).

Aceştia preced specificatorii de format amintiţi în secţiunea anterioară. În situaţia în care apar, sunt situaţi imediat după semnul `%` şi înainte de litera specificator de format utilizată.

Exemple: `%hd, %ld, %lf, %Fp`

Modificatorul `h` se poate aplica oricărui specificator asociat tipurilor N şi Z. Efectul este interpretarea datelor drept întregi `short`.

Observaţii:

1. După cum a fost amintit, un tip `short int` reprezintă tipul întreg implicit în momentul declarării întregilor folosind `int`. Se omite utilizarea lui `short`. Acest tip de dată (`int`) poate avea rezervaţi 2B sau 4B, depinzând de compilatorul concret utilizat.
2. Pentru cazul în care se doresc întregi interpretaţi în mod expres întregi fără semn (şi numai ei, numerele **R** având rezervat obligatoriu un bit pentru

semn) se poate folosi particula `u`, care *precede* specificatorul întreg concret dorit.
Exemplu: `%ui` înseamnă `unsigned int`.

Modificatorul `l` poate fi utilizat atât în conjuncție cu tipurile întregi cât şi cu cele reale. În cazul celor *întregi* poate fi aplicat oricărui specificator asociat tipurilor întregi (`d, i, o, O, u, x, X`), efectul fiind dublarea spațiului de memorie față de cazul short. Tipul de dată construit este `long int`. Aceeaşi observație ca şi în cazul modificatorului `h` aplicat întregilor: spațiul exact alocat depinde de compilator.

Pentru *numerele reale* acesta se poate aplica specificatorilor: `e, E, f, g, G`. Tipul real astfel construit este cel dublă precizie: `double`.

Modificatorul `L` este dedicat exclusiv familiei **R**. Aplicat asupra unuia dintre specificatorii `e, E, f, g, G` are ca efect construirea tipului de dată real *precizie extinsă* (`long double`).

Mai există doi modificatori de format speciali: `F` şi `N`. Aceştia sunt dedicați pointerilor şi şirurilor de caractere. Pot fi aplicați doar specificatorilor: `p`, `n` şi `s`. Adică doar entităților de *tip adresă*.

Pentru `F` semnificația este de `far` (*inter-segmente*), iar pentru `N` de `near` (*intra-segment, în cadrul aceluiaşi segment de memorie*). Este vorba de o discuție în contextul modurilor de adresare, frecvent întâlnite de exemplu în *limbaj de asamblare*. Situațiile în care aceşti modificatori sunt utili țin de *modelul de memorie* utilizat la compilare. Există modele de memorie pentru care una sau mai multe dintre zonele componente ale programului din memorie (*cod, date, stivă*) ocupă mai mult de 1 *segment* (segmentul este definit ca o zonă continuă de memorie de dimensiune 64KB). Un exemplu de model de memorie este cel *large*. Aici zona alocată codului, textului şi stivei poate depăşi 1 segment. Pentru această situație este util modificatorul `F`. Pentru celelalte situații, în care spațiul de memorie asociat uneia dintre componentele amintite ale programului nu depăşeşte 1 segment este folosit modificatorul `N` (eventualele adresări se efectuează în cadrul aceluiaşi segment, deci în limita a 64KB).

5. UN EXEMPLU CONCRET DE PROGRAM *C*

Pe baza celor prezentate până în acest moment, iată un *program C complet*.

Exemplul ales este cel pentru calculul numărului `ab`, unde `a` şi `b` sunt numere naturale, putând lua şi valoarea 0.

În C funcția care realizează această operație este:

```
pow(baza, exponent)
```

care permite ca atât baza cât şi exponentul să fie numere reale. Prototipul său este:

```
double pow(double, double);
```

şi se află în <math.h>. Argumentele sunt convertite automat la tipul double.

Prima variantă care ne vine în minte este următoarea:

```
unsigned long putere(      unsigned long a,
                           unsigned long b)
{
 int i;  // contor pentru bucla for() ce urmează.
 unsigned long rez = a;    // definesc variabila 'rez'
         // şi o iniţializez cu valoarea bazei. Acesta
         // este o variablă locală funcţiei, nefiind
         // cunoscută de nici o altă funcţie din fişierul
         // în care va apărea această funcţie. Funcţia
         // main() nu face excepţie.

  if(a==0) return 0; // dacă baza este nulă, atunci
         // returnez 0.
    if(b==0) return 1;     // dacă exponentul este 0, atunci
         // returnez 1.
    else if(b==1) return a;
          else for(i=1; i<b; i++) rez *= a;

    return rez;
   }
```

Am ales atât baza cât şi exponentul de tip long (pe 4 octeţi) pentru a asigura o generalitate mai mare a algoritmului. Dacă am fi ales întregul de tip short (pe 2 octeţi), în cazul unor numere cea ar fi depăşit această gamă am fi înregistrat depăşiri a capacităţii de memorare pentru întregi. Bineînţeles că în practică nu vom ajunge la operaţii atât de bizare precum 12000^{123} . Numărul de operaţii în această variantă nu este cel mai mic posibil, fiind egal cu b-1. Dar în cazul unor puteri mari, acest număr de operaţii creşte în timp polinomial. Algoritmul este de ordinul *O(b)*.

Un algoritm *optim* trebuie să aibă numărul cel mai mic de operaţii posibil. De aceea se adoptă următoarea variantă, în care numărul de operaţii este mai mic decât în cazul anterior.

Calculul se bazează pe formula:

$$a^b = \begin{cases} (a^{b/2})^2 & , \quad \text{pentru b par.} \\ a*(a^{(b-1)/2})^2 & , \quad \text{pentru b impar.} \end{cases}$$

Varianta este:

```
   unsigned long putere1(  unsigned long a,
                              unsigned long b)
   {
    unsigned long rez = 1;
    while(b>0){      // while().
         if(!b%2) {        // pentru b par.
                     a = a*a;
```

```
                    b /= 2;
                    }
      else {              // pentru b impar.
            b -= 1;
            rez *= a;
            }
 }// while().
return rez;
}
```

Programul complet următor este scris folosind a doua variantă.

```
#include<stdio.h>
#include<conio.h>
unsigned long putere1(unsigned long, unsigned long);
      // prototipul funcţiei, adică declaraţia ei. Aici
      // compilatorul nu este interesat de numele
      // argmentelor. Chiar dacă sunt date ele sunt
      // ignorate.
int main()
{
 unsigned long baza, exponent;
 printf("\n Care este baza ?(număr natural)");
 scanf("%uld", &baza);  // modificatorul '%ld'
                        // înseamnă long integer
 printf("\n Care este exponentul?(număr natural)");
 scanf("%uld", &exponent);

 printf("\n\n Rezultatul este %uli.",
                              putere1(baza, exponent));
 scanf("%u", &baza); // opreşte ecranul.
 return 0;
}
// după corpul funcţiei main() este definită funcţia
// utilizator, adică se stabileşte corpul acesteia.
unsigned long putere1(  unsigned long a,
                           unsigned long b)
{
 unsigned long rez = 1;

 while(b>0){
         if(!b%2) {     // pentru b par.
                     a = a*a;
                     b /= 2;
                  }
      else { // pentru b impar.
            b -= 1;
            rez *= a;
               }
    }
   return rez;
}
```

Algoritmul este foarte interesant: în caz că puterea nu este număr par, se scade din ea o unitate, făcând în acelaşi timp înmulțirea cu `baza`. Deci la fiecare scădere cu o unitate a exponentului se face şi o înmulțire cu `baza` a noului număr. Este normal ca pe măsură ce puterea scade cu o unitate, aceasta să se reflecte în rezultat printr-o înmulțire cu `baza`. Orice putere multiplu de 2 se reflectă în rezultat printr-o înmulțire cu `baza*baza`, atâta timp cât valoarea acelei puteri este strict pozitivă.

Specificatorii de format '`%uld`' şi '`%uli`' semnifică un *întreg lung fără semn*. Pentru lămuriri consultați ***Secțiunea 4 - Lista specificatorilor de format şi a modificatorilor acestuia*** din Anexa de față.