1

Funcții definite în program

O funcţie furnizează o modalitate convenabilă de a încapsula anumite calcule care pot fi utilizate (apelate) ulterior fără a ne preocupa, în momentul apelului, de modul lor de implementare.

Limbajul de programare C face folosirea funcţiilor uşoară, convenabilă şi eficientă.

Definiția unei funcţii are următoarea formă:

```
tipul-rezultatului numele-functiei(declaratiile-parametrilor)
  {
      declaratii
      instructiuni
  }
```

Programul 1.1: Definirea şi utilizarea unei funcţii.

```
#include <stdio.h>
int putere(int m, int n);
 /* utilizarea functiei putere */

int main()
{
   int i;
   for (i=0; i<10; ++i)
      printf("%d %d %d\n", i, putere(2,i), putere(-3,i));
   return 0;
}

/* putere: ridica baza la puterea n; n>=0 */
int putere(int baza, int n)
{
    int i,p;
    p=1;
    for (i=1; i<=n; ++i)
        p = p * baza;
    return p;
}
```

Prima linie a funcţiei

```
int putere (int baza, int n)
```

declară *tipurile şi numele parametrilor* precum şi *tipul rezultatului* returnat de funcţie.

În general, vom folosi denumirea de *parametru* pentru o variabilă enumerată în lista închisă între paranteze din definiţia unei funcţii şi cea de *argument* pentru valoarea folosită într-un apel de funcţie.

Valoarea calculată în funcţia `putere` este returnată către `main` de instrucţiunea `return`. După `return` poate urma orice expresie. Nu este obigatoriu ca o funcţie să returneze

o valoare. Ea poate fi apelată şi pentru efectele directe pe care le produce (*de exemplu* o funcție care afişează pe ecran o valoare pe care o primeşte ca argument).

Declarația

```
int putere(int m, int n);
```

aflată înaintea lui main se numeşte *prototip de funcție* şi trebuie să concorde cu definiția şi apelurile funcției `putere`. Nu este necesar ca numele parametrilor să corespundă. Un prototip poate fi scris şi omițând numele parametrilor:

```
int putere(int, int);
```

Numele folosite în definiția efectivă a funcției `putere` pentru parametrii săi sunt *entități locale* funcției `putere` şi nu sunt vizibile pentru nici o altă funcție. Acest lucru este valabil şi pentru variabilele locale `i` şi `p`. Aceste variabile se numesc, în C, *variabile automatice.*

În C, toate argumentele funcțiilor sunt transmise "*prin valoare*". Acest lucru înseamnă că funcția apelată primeşte valorile argumentelor sale prin stocare în variabile temporare create în stivă special în acest scop. În acest fel, *funcția nu are acces la locațiile de memorie ale variabilelor originale*. În consecință, *funcția apelată nu poate modifica direct variabila corespunzătoare argumentului transmis din funcția apelantă*; ea nu poate modifica decât copia temporară.

Apelul prin valoare conduce la programe mai compacte, cu mai puține variabile neesențiale, deoarece parametrii pot fi tratați în rutina apelată ca variabile locale, convenabil inițializate ca urmare a apelului.

Când este necesar, se poate face ca o funcție să modifice o variabilă din funcția apelantă. Funcția apelantă trebuie să furnizeze *adresa variabilei* ce va fi accesată (un *pointer* către acea variabilă), iar funcția apelată trebuie să declare *parametrul ca fiind pointer* şi să acceseze variabila *indirect*, prin intermediul acestuia (vezi cap. 2 şi cap.4).

O altă excepție este în cazul tablourilor: când numele unui tablou este folosit ca argument, valoarea transmisă funcției este adresa locației primului element al tabloului. Folosind această valoare ca pe o variabilă cu indici, funcția poate accesa şi poate modifica *direct* orice element al tabloului vezi şi cap.2.

1.1 Bazele definirii şi utilizării funcțiilor

Există două mari *avantaje* ale utilizării funcțiilor definite în program:

1) împărțirea unor sarcini de calcul ample în unele mai mici, ceea ce conduce la scrierea mai simplă a programului în ansamblu;

2) reutilizarea unui cod deja scris sub formă de funcții pentru programe anterioare (*de exemplu* funcțiile de bibliotecă).

Limbajul de programare C a fost special conceput să permită utilizarea simplă şi eficientă a funcțiilor; programele C sunt, în general, formate din multe funcții mici şi nu din câteva funcții mari.

Programul 1.2: Să se scrie un program care tipăreşte liniile din fişierul de intrare ce conțin un anumit "şablon" (şir de caractere prestabilit).

Algoritmul problemei se împarte în trei părți:

```
while (mai exista o linie)
    if (linia contine sablonul)
        tipareste-o
```

Deşi ar fi posibil să plasăm întregul cod în funcția `main`, o soluție mai bună este să facem din fiecare parte a algoritmului o funcție separată (funcțiile `preialinie`, `indicesir` şi respectiv `printf` – funcție de bibliotecă).

Funcția `indicesir(s,t)` returnează *poziția* sau indicele din şirul `s` unde începe şirul `t` sau -1 dacă `s` nu îl conține pe `t`. Deoarece în C tablourile încep de la poziția 0, indicii vor fi *zero* sau strict pozitivi, o valoare negativă precum -1 fiind convenabilă pentru semnalarea eşecului.

```
#include <stdio.h>
#define MAXLINIE 1000 /* lung. max. a liniei de intrare */
int preialinie(char linie[ ], int max);
int indicesir(char sursa[ ], char decautat[ ]);
char sablon[ ] = "ea";   /* sablonul cautat */
/* gaseste toate liniile in care apare sablonul */

int main()
{
   char linie[MAXLINIE];
   int gasite = 0;
   while (preialinie(linie, MAXLINIE) > 0)
      if (indicesir(linie, sablon) >= 0)  {
         printf("%s", linie);
         gasite++;   }
   return gasite;
}

/* preialinie: preia linia, o copiaza in s, returneaza lung.*/
int preialinie(char s[ ], int lim)
{
   int c, i;
   i = 0;
   while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
       s[ i++ ] = c;
   if (c == '\n')
       s[ i++ ] = c;
   s[ i ] = '\0';
   return i;
}

/* indicesir: returneaza indicele lui t din s sau -1 daca t
     nu exista in s*/
int indicesir(char s[ ], char t[ ])
{
   int i, j, k;
   for (i = 0; s[ i ] != '\0'; i++)  {
      for (j=i, k=0; t[ k ]!='\0' && s[ j ]==t[ k ]; j++, k++) ;
      if (k > 0 && t[ k ] == '\0')
         return i;
   }
   return -1;
}
```

Revenind acum la forma generală a definiției unei funcții:

```
tip-rezultat nume-functie(declaratii de parametri)
{
   declaratii si instructiuni
}
```

anumite părți pot lipsi; o funcție minimală este:

```
nimic() { }
```

care nu face nimic şi nu returnează nimic. O astfel de formă poate fi utilă ca variantă provizorie pe parcursul dezvoltării programului.

Dacă tipul returnat este omis se presupune că este `int`.

Un program este o colecție de definiții de variabile şi de funcții. *Comunicarea între funcțiile* unui program se efectuează prin:

- parametri (argumente);
- valori returnate;
- variabile externe.

Funcțiile pot apărea în orice ordine în fişierul sursă, iar programul sursă poate fi împărțit în mai multe fişiere, atât timp cât nici o funcție nu este divizată.

Mecanismul pentru *returnarea unei valori* din funcția apelată către apelanta sa (rezultatul funcției) este instrucțiunea `return` .

După `return` poate urma orice expresie :

```
return expresie;
```

Dacă este necesar, `expresie` poate fi convertită la tipul returnat de funcție. De asemenea, pentru claritate, `expresie` poate fi inclusă, opțional, între paranteze. Dacă nu are nevoie de ea, funcția apelantă poate să ignore valoarea returnată.

După instrucțiunea `return` nu este necesar să urmeze vreo expresie; caz în care nu se returnează nici o valoare apelantei. În această situație, apelul funcției se justifică doar prin efectele colaterale pe care le determină: *de exemplu*, afişarea unei valori. Este o situație similară cu aceea a unei proceduri Pascal.

Dacă se ajunge la acolada închisă `}` care indică terminarea funcției, controlul revine de asemenea la apelantă fără să transmită nici o valoare. Nu este ilegal, dar poate fi un semnal de alarmă pentru o posibilă greşală, dacă o funcție returnează o valoare într-un caz şi nu returnează nici una în alt caz.

Procedeul pentru compilarea şi încărcarea unui program C care se află în *mai multe fişiere sursă* variază de la un sistem la altul. El va fi prezentat în alt capitol.

Programul 1.3: Să se determine maximul dintre trei numere întregi, utilizând o funcție care determină maximul dintre două numere întregi.

```
#include <stdio.h>
int max (int a,int b)
{
   if (a > b)
      return a;
   else
      return b;
}
```

```
main( )
{
   int x, y, z;
   printf("Introduceti 3 numere intregi\n");
   scanf("%d%d%d", &x, &y, &z);
   printf("Maximul este %d\n", max(x, max(y,z));
}
```

1.2 Funcții ce returnează altfel de valori decât întregi

Exemplele de funcții de până acum, fie nu au returnat nici o valoare fie au returnat un `int`.

Ce se întâmplă dacă o funcție trebuie să returneze un rezultat de alt tip?

1. *Funcția trebuie să declare tipul valorii pe care o returnează*, din moment ce aceasta nu este `int`. Numele tipului precede numele funcției.
2. *În rutina apelantă sau înainte de codul apelantului se declară explicit funcția apelată* întrucât rutina apelantă trebuie să ştie că funcția apelată returnează o valoare ce nu aparține tipului `int`.

Programul 1.4: Să se scrie o nouă variantă a funcției `putere` care calculează baza $^{\text{exponent}}$, unde `baza` este o valoare reală iar `exponent` este un număr natural.

```
#include <stdio.h>
float putere (float baza, int exponent)
{
   float p;
   int i;
   if (exponent==0) return 1;
   for (i=1, p=1.0; i<=exponent; i++)
      p*=baza;
   return p;
}
main()
{
   int e;
   float b, p;
   printf("Introduceti baza si exponentul \n");
   scanf("%f%d", &b, &e);
   p=putere(b,e);
   printf("%f la puterea %d = %f\n", b, e, p);
   printf("5 la puterea 3 = %f\n", putere(5,3));
   printf("(2 la 3) la 2 = %f\n", putere(putere(2,3),2));
}
```

Programul 1.5: Scrierea şi utilizarea funcției `atof` care converteşte şirul de caractere `s` în numărul echivalent în reprezentare virgulă mobilă în dublă precizie.
Biblioteca standard include o funcție `atof`; este declarată în fişierul antet `<stdlib.h>`.

```
#include <ctype.h>
/* atof: converteste sirul de caractere s la tipul double */
double atof(char s[ ])
{
   double val, putere;
   int i, semn;
```

```
   for (i=0; isspace(s[i]); i++) /* treci peste spatiile albe */
      ;
   semn = (s[i] == '-') ? -1 : 1;
   if (s[i] == '+'|| s[i] == '-')
      i++;
   for (val = 0.0; isdigit(s[i]); i++)
      val = 10.0 * val + (s[i] -'0');
   if (s[i] == '.')
      i++;
   for (putere = 1.0; isdigit(s[i]); i++) {
      val = 10.0 * val + (s[i] -'0');
      putere *= 10.0;
   }
   return semn * val / putere;
}
```

Rutina apelantă este un calculator rudimentar care citeşte numerele câte unul pe linie, precedate opțional de un semn, le adună şi tipăreşte suma curentă după fiecare număr introdus.

```
#include <stdio.h>
#define MAXLINIE 100
/* calculator rudimentar */
main()
{
   double suma, atof(char [ ]);
   char linie[MAXLINIE];
   int preialinie(char linie[ ], int max);
   suma = 0.0;
   while (preialinie(linie, MAXLINIE) > 0)
       printf("\t%g\n", suma += atof(linie));
   return 0;
}
```

Declarația

```
double suma, atof(char [ ]);
```

precizează că suma este o variabilă de tip `double` şi că `atof` este o funcție care primeşte un argument de tip `char[]` şi returnează un double.

Funcția `atof` trebuie *declarată, definită şi utilizată în mod concordant*. Dacă funcția `atof` şi apelul său din `main` au, în acelaşi fişier, tipuri care nu concordă, eroarea va fi detectată de compilator. În schimb, dacă funcția `atof` ar fi compilată separat (ceea ce este posibil) neconcordanța nu ar fi detectată. Funcția `atof` ar returna un `double` pe care funcția `main` l-ar trata, de exemplu, ca pe un `int` şi ar apărea rezultate lipsite de înțeles.

Dacă o funcție nu are prototip, aceasta este *declarată implicit* de prima sa apariție într-o expresie, cum ar fi:

```
suma += atof(linie)
```

Dacă un nume care nu fost declarat în prealabil apare într-o expresie, urmat de o paranteză deschisă, acesta este declarat prin context ca fiind nume de funcție, despre funcție se presupune că este de tip `int`, iar despre argumentele acesteia nu se presupune nimic. De asemenea, dacă o declarație de funcție nu include argumente, ca de exemplu:

```
double atof();
```

atunci acest lucru este de asemenea interpretat ca însemnând că nu se poate deduce nimic despre argumentele funcției `atof` şi, în consecință, nu se efectuează nici o verificare asupra parametrilor.

Se recomandă ca, într-o declarație de funcție, dacă funcția acceptă argumente, să declarăm parametrii corespunzători sau să folosim tipul `void` dacă funcția nu acceptă argumente.

Programul 1.6: Realizarea funcției `atoi` prin apelul funcției `atof` şi conversia explicită a rezultatului.

```
/* atoi: converteste sirul s la un intreg folosind atof */
int atoi(char s[ ])
{
   double atof(char s[ ]);
   return (int) atof(s);
}
```

Valoarea returnată din primul apel este convertită la tipul funcției înainte de a se face returnarea din noua funcție. Operatorul *cast* precizează că operația este intenționată şi elimină orice avertisment din partea compilatorului.

1.3 Variabile externe

Un program C este format dintr-o colecție de obiecte externe care sunt variabile sau funcții. Adjectivul "*extern*" este folosit ca antonim pentru "intern" care descrie argumentele şi variabilele definite în interiorul funcțiilor.

Variabilele externe sunt *definite în afara oricărei funcții* şi sunt astfel *disponibile mai multor funcții*.

Funcțiile în sine sunt întotdeauna externe, deoarece *limbajul C nu permite definirea funcțiilor în interiorul altor funcții*.

Variabilele şi funcțiile externe au proprietatea că toate trimiterile către ele prin acelaşi nume, chiar şi din funcții compilate separat, constituie trimiteri către acelaşi obiect (standardul numeşte această proprietatea "*legare externă*"). Limbajul de programare C permite totu□i să se definească variabile şi funcții externe care să fie vizibile doar dintr-un singur fişier sursă.

Deoarece variabilele externe sunt *global accesibile* ele pot constitui o alternativă pentru *transmiterea datelor între funcții*, prin opoziție cu argumentele funcțiilor şi cu valorile returnate de acestea. Orice funcție poate accesa o variabilă externă făcând trimitere la aceasta prin nume, dacă numele a fost declarat în vreun fel.

Dacă mai multe funcții urmează să folosească în comun un număr mare de variabile, variabilele externe sunt mai convenabile pentru a reprezenta aceste variabile, decât listele de argumente lungi. Acest lucru trebuie folosit cu precauție pentru că poate avea efect dăunător asupra structurii programului fiind prea multe conexiuni de date între funcții.

Variabilele externe sunt utile şi datorită *domeniului de vizibilitate şi duratei de existență mai mari.* Aceste variabilele au *caracter permanent*. Ele păstrează valorile de la o invocare de funcție la următoarea, spre deosebire de variabilele *automatice* care sunt interne unei funcții (sunt create când se intră în funcție şi dispar când se iese din aceasta).

Astfel, *dacă două funcții trebuie să folosească în comun anumite date, dar nici una dintre ele nu o apelează pe cealaltă, este adesea convenabil ca datele comune să fie păstrate în variabile externe* în loc să fie transmise şi preluate prin intermediul argumentelor.

1.4 Reguli pentru domeniul de vizibilitate

Domeniul de vizibilitate al unui nume este *partea programului în care acesta poate fi folosit*, în conformitate cu declarația sau definiția sa.

Pentru *o variabilă automatică* declarată la începutul unei funcții, domeniul de vizibilitate este *funcția în care este declarat numele*. Variabilele locale cu acelaşi nume din diferite funcții nu au legătură unele cu altele. Acelaşi lucru este valabil şi pentru parametrii funcției, care sunt de fapt variabile locale.

Domeniul de vizibilitate al unei *variabile sau funcții externe începe din locul în care aceasta este declarată şi ține până la sfârşitul fişierului în curs de compilare*.

```
main()  { … }
int i = 0;
double tab[MAX];

void func1(double f)  { … }

double func2(void) { … }
```

Variabilele `i` şi `tab` pot să fie folosite în funcțiile `func1` şi `func2` prin numirea lor, fără să mai fie nevoie de alte declarații. Ele nu sunt vizibile în `main`; la fel, nu sunt vizibile în `main` nici funcțiile `func1` şi `func2`.

Dacă trebuie să se facă referire la o variabilă externă înainte ca aceasta să fie definită, sau dacă este definită într-un fişier sursă diferit de cel în care este folosită, atunci este obligatorie o declarație `extern`.

Care este diferența între *declarația* unei variabile şi *definiția* sa?

O **declarație** anunță *proprietățile* unei variabile (în primul rând tipul său).

O **definiție** anunță *proprietățile* unei variabile şi, în plus, produce *alocarea de spațiu* de memorie pentru stocarea valorii variabilei.

Exemple:

1. Liniile

```
int i = 0;
double tab[MAX];
```

din exemplul anterior, apar în afara oricărei funcții: sunt *definiții* de variabile externe (`i` şi `tab`).

2. În schimb, liniile

```
extern int i;
extern double tab[ ];
```

declară pentru restul fişierului sursă că `i` este un `int` şi că `tab` este un tablou de tip `double`, dar nu crează variabilele şi nu alocă spațiu de stocare pentru acestea.

Trebuie să existe o *singură definiție* a unei variabile externe, doar într-unul din fişierele care compun programul sursă. Pentru a accesa o astfel de variabilă în alte fişiere, aceste fişiere trebuie să conțină declarații `extern` corespunzătoare.

Dimensiunile tablourilor trebuie specificate la definire. Ele sunt opționale în cazul declarațiilor `extern`. De asemenea, *inițializarea* unei variabile externe se poate face *doar la definire,* adică acolo unde ea nu este declarată `extern` şi are rezervată locația de memorie în care să se înregistreze această valoare.

1.5 Variabile statice

Declarația (clasa de memorare) `static` aplicată unui *obiect extern* (variabilă sau funcție), limitează domeniul de vizibilitate al acelui obiect la restul fişierului în curs de compilare. Declarația `static` face ca o variabilă externă să nu poată fi accesată de funcții din afara fişierului în curs de compilare.

Declarația `static` externă este utilizată cel mai adesea pentru variabile, dar poate fi aplicată şi funcțiilor, cu acelaşi efect. De obicei, numele de funcții sunt globale (sunt vizibile pentru orice parte a programului). Dacă o funcție este declarată ca fiind `static`, numele acesteia nu este vizibil în afara fişierului în care este declarată.

Declarația `static` poate fi aplicată, de asemenea, şi *variabilelor interne*. Variabilele interne de tip `static` sunt locale unei anumite funcții, la fel ca şi variabilele *automatice*, dar ele există în permanență. Nu sunt create şi nici nu sunt distruse de fiecare dată când este activată funcția. Variabilele interne de tip `static` au spațiu de stocare permanent în cadrul unei funcții şi, în consecință, pot fi folosite pentru *transmiterea de valori de la un apel la altul al aceleiaşi funcții.*

Clasa de memorie `static` este precizată prefixând declarația normală prin cuvântul `static`.

Dacă într-un fişierul ce conține funcțiile `func1` şi `func2` declarăm `static` variabilele `i` şi `tab`:

```
static int i = 0;
static double tab[MAX];

void func1(double f)  { … }
double func2(void) { … }
```

atunci variabilele `i` şi `tab` nu pot fi accesate de nici o altă funcție din alte fişiere sursă, doar de `func1` şi `func2`.

1.6 Statutul variabilelor - sinteză

Variabilele **interne** unei funcții: sunt vizibile doar în interiorul funcției respective. Din punctul de vedere al duratei de viață pot fi:

- *automatice* - sunt create când se intră în funcție şi dispar când se revine din aceasta;

- *statice* - există pe toată durata programului; nu sunt create şi nici nu sunt distruse la fiecare activare funcției; în consecință, pot fi folosite pentru transmiterea de valori de la un apel la altul al aceleiaşi funcții.

În mod *implicit*, variabilele interne sunt *automatice.* Pentru a deveni *statice*, ele trebuie declarate `static`.

Variabilele **externe** tuturor funcțiilor: sunt vizibile cel puțin până la sfârşitul fişierului sursă respectiv. Variabilele externe au *caracter permanent*: locațiile lor de memorie există pe toată durata execuției programului. Rezultă că ele pot fi folosite pentru transmiterea de date între apelurile de funcții diferite.

Variabilele externe pot fi declarate:

- `static` – domeniul de vizibilitate se limitează la fişierul în curs de compilare;
- `extern` – *declară* proprietățile unei variabile *definită* într-un alt fişier sursă; unica locație de memorie pentru variabilă va fi în fişierul în care ea este definită (în care nu este extern).

Declarația `static` se poate aplica şi funcțiilor, cu acelaşi efect privind vizibilitatea.

1.7 Variabile registru

O declarație `register` avertizează compilatorul că variabila în discuție va fi intens folosită. Variabilele de tip `register` se păstrează în registrele maşinii (care se găsesc chiar în interiorul unității centrale de prelucrare) şi fac programul mai performant.

Compilatorul poate să acceseze o valoare mult mai repede atunci când ea este localizată într-un registru. Pentru că numărul de registre este limitat, compilatorul nu poate să asocieze permanent o variabilă unui registru, dar poate să încerce să țină variabila cât mai mult în registru.

Exemple:

```
register int x;
register char c;
```

Cuvântul `register` este ignorat în cazul declarațiilor în exces.

Declarația register nu poate fi aplicată decât entităților locale: variabile automatice şi parametrii unei funcții.

Exemplu:

```
f(register unsigned m, register long n)
{
    register int i;
    ….
}
```

Restricțiile referitoare la numărul şi tipurile posibile pentru *variabile registru* diferă de la maşină la maşină.

1.8 Structura de bloc

C-ul nu este un limbaj structurat pe blocuri ca şi Pascal-ul, deoarece funcţiile nu pot fi definite în interiorul altor funcţii. În schimb, variabilele pot fi definite în maniera unei structuri în blocuri în interiorul unei funcţii.

Declaraţiile de variabile pot fi plasate după acolada deschisă care introduce *orice* instrucţiune compusă, nu numai după acolada care marchează începutul unei funcţii. Domeniul de vizibilitate al variabilei este cuprins între cele două acolade ({ … }).

O variabilă *automatică* declarată şi iniţializată într-un bloc este iniţializată de fiecare dată când se intră în bloc.

Exemplu:

```
if (n>0) {
            int i=0;  /* declara şi initializeaza pe i la
                         fiecare parcurgere*/
...
}
```

O variabilă declarată *static* este *iniţializată numai prima dată* când se intră în bloc.

1.9 Iniţializarea variabilelor

În absenţa iniţializării explicite este garantată iniţializarea cu zero doar a variabilelor *externe* şi *statice*. Variabilele *automatice* şi *registru* au *valori* iniţiale *nedefinite*.

Pentru variabilele externe şi statice, valoarea de iniţializare explicită trebuie să fie o expresie constantă; *iniţializarea se efectuează o singură dată*, în principiu înainte ca programul să înceapă execuţia.

În cazul variabilelor automatice şi registru, iniţializarea se efectuează de fiecare dată când se intră în funcţie sau în bloc. Valoarea de iniţializare poate fi şi o expresie care să implice valori definite anterior, chiar şi apeluri de funcţii.

Exemplu:

```
int cautbin(int x, int v[ ], int n)  {
    int prim = 0;
    int ultim = n – 1;
    int mijl;
    ...
}
```

Un tablou se poate iniţializa punând după declaraţia sa o listă de valori de iniţializare:

```
int zile[]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Când este omisă dimensiunea, compilatorul va calcula lungimea numărând valorile de iniţializare (12 în acest caz).

Dacă sunt mai puţine valori decât dimensiunea specificată, elementele lipsă vor fi zero atât pentru variabilele externe sau statice cât şi pentru cele automatice. Situaţia inversă: prea multe valori de iniţializare, constituie o eroare.

Tablourile de caractere se pot iniţializa cu un şir de caractere, astfel:

```
char luna[ ]="iunie";
```

în loc de

```
char luna[ ]={'i','u','n','i','e','\0'};
```

Dimensiunea tabloului în acest caz este 6 (5 caractere plus terminatorul '\0').

Programul 1.7: Evaluarea unei funcţii intr-un punct.

```
/*Calculul valorii unei functii date intr-un punct
f(x) = |x^2,      x <= -1
       |x+3,      -1 < x <= 1
       |arctg(x),  x > 1 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
float f(float x)  {
   if (x <= -1) return x*x;
   else if (x <= 1) return x+3;
          else return atan(x);   }

int main(void) {
   float v;
   printf("Calculul valorii unei functii pentru ");
   printf("un argument citit de la tastatura  \n");
   printf("Valoarea argumentului este:");
   scanf("%f",&v);
   printf("Valoarea functiei in punctul %.2f este%.2f \n",v,f(v));
   system("pause");
   return 0;
}
```

Programul 1.8: Algoritmul lui Euclid de aflare a celui mai mare divizor comun a două numere întregi.
Formularea în cuvinte a algoritmului este următoarea:
Dacă unul dintre numere este zero, c.m.m.d.c. al lor este celălalt număr.
Dacă nici unul dintre numere nu este zero, atunci c.m.m.d.c. nu se modifică dacă se înlocuieşte unul dintre numere cu restul împărţirii sale cu celălalt.

```
/*determinarea cmmdc si cmmmc pentru doua numere intregi*/
#include <stdio.h>
int cmmdc(int m, int n) {
    int r;
    r=m % n;
    while (r != 0) {
       m=n;
       n=r;
       r=m % n;
    }
    return n;
}
```

```
int main(void) {
   int n1,n2,divizor,multiplu;
   printf("Calculul cmmdc si cmmmc cu alg. lui Euclid\n");
   printf("Numerele sunt:\n");
   printf("n1 = ");  scanf("%d",&n1);
   printf("n2 = ");  scanf("%d",&n2);
   divizor=cmmdc(n1,n2);
   printf("CMMDC = %d\n",divizor);
   printf("CMMMC = %d\n",(n1*n2)/ divizor);
   return 0;
}
```

1.10 Întrebări şi probleme

1. Care sunt *avantajele utilizării funcţiilor* definite în program?
2. Care este forma generală a *definiţiei unei funcţii* ?
3. Care este *tipul returnat implicit* ?
4. Cum se poate face *comunicarea între funcţii* într-un program C ?
5. Care este sintaxa (modul de redactare) şi semantica (efectul) *instrucţiunii* `return` ?
6. Ce sunt *variabilele externe* ? Dar *variabilele declarate ca* `extern` ?
7. În ce constă proprietatea de *legare externă* ?
8. La ce pot fi *utilizate variabilele externe* ?
9. Ce sunt *variabilele automatice* ?
10. Ce este *domeniul de vizibilitate* al unui nume ? Dar *durata de viaţă* a unei variabile ?
11. Care este diferenţa între *declaraţia* unei variabile şi *definiţia* sa?
12. Ce efect are declaraţia `static` pentru o *variabilă externă* ? Dar pentru una *internă* ?
13. Ce efect are o declaraţie `register` ?
14. Ce înseamnă *bloc* în limbajul de programare C ?
15. Cum *se iniţializează variabilele simple* în limbajul de programare C ? Dar *tablourile* ?
16. Să se citească două numere întregi pozitive, `n` şi `m`. Să se calculeze, cu ajutorul a două funcţii `suma(n)` şi `produs(m)` valoarea expresiei

$$E=\sum_{i=1}^{n}\frac{2i}{3i^2+4}+\sqrt{\prod_{j=1}^{m}(j^2+1)}$$

17. Să se citească mai multe numere naturale. Să se calculeze, pentru fiecare număr, cu ajutorul unei funcţii, suma cifrelor şi să se afişeze.
18. Un număr perfect este egal cu suma divizorilor săi, printre care este considerată valoarea 1, dar nu şi numărul. Să se găsească, cu ajutorul unei funcţii, toate numerele perfecte mai mici sau egale cu un număr dat n şi să se afişeze fiecare număr astfel determinat, urmat de suma divizorilor săi. de exemplu, numărul 6 are divizorii 1, 2 şi 3 şi este număr perfect deoarece 6=1+2+3.

19. Să se scrie o funcție care verfică dacă cifra `c` aparține sau nu reprezentării zecimale a numărului. Să se apeleze această funcție pentru a determina cifrele comune ale numerelor `n` şi n^p. Valorile `n` şi `p` sunt date de intrare (numere întregi).

20. Să se citească un număr natural `n`. Să se afişeze toate palindroamele mai mici sau egale cu `n`. Un număr natural se numeşte *palindrom* dacă are aceeaşi valoare indiferent dacă cifrele sale sunt citite de la stânga la dreapta sau de la dreapta la stânga. Să se scrie o funcție care testează dacă parametrul ei este palindrom.

21. Fie `x` un număr natural, `x>2`. Să se scrie programul care determină cel mai mare număr prim mai mic decât `x` şi cel mai mic număr prim mai mare decât `x`.

22. Fie `n` un număr natural. Să se determine, prin intermediul unei funcții, toate numerele naturale mai mici decât `n` cu proprietatea că sunt divizibile atât prin suma, cât şi prin produsul cifelor sale.

23. Fie `n` un număr natural şi `x` un număr real, citite de la tastatură. Să se calculeze, cu ajutorul unei funcții, valoarea expresiei:

$$a.\quad E(n,x)=1+\frac{x}{1!}+\frac{x^2}{2!}+\ldots+\frac{x^n}{n!}$$

$$b.\quad E(n,x)=\frac{x}{1!}-\frac{x^3}{3!}+\frac{x^5}{5!}-\ldots(-1)^n\frac{x^{2n+1}}{(2n+1)!}$$

$$c.\quad E(n,x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}+\ldots+(-1)^n\frac{x^{2n}}{(2n)!}$$

24. Să se citească `n` numere întregi, pe rând într-o aceeaşi variabilă. Să se calculeze suma numerelor prime. Să se utilizeze în program o funcție care determină dacă un număr natural este prim.

25. Fiind dat un număr natural `n`, să se determine toate numerele naturale mai mici decât `n`, cu proprietatea că suma cuburilor cifrelor lor este egală cu numărul însuşi. Să se scrie o funcție care calculează suma cuburilor cifrelor unui număr întreg.

26. Să se citească trei numere reale pozitive. Să se determine, cu ajutorul unei funcții, dacă pot reprezenta laturile unui triunghi. Dacă da, să se calculeze perimetrul şi aria triunghiului.

27. Se dă un şir de numere naturale, citite pe rând într-o aceeaşi variabilă. Să se determine, cu ajutorul unei funcții, câte elemente sunt pătrate perfecte.

28. Să se scrie programul care, citind o listă de numere întregi, afişează numărul prim cel mai apropiat de fiecare element al listei. Dacă două numere prime sunt la distanță egală de un element, se vor afişa ambele. Să se scrie o funcție care determină dacă un număr este prim.