6. FUNCȚII

6.1. Rolul si modul de funcționare ale unei funcții

Pentru a rezolva unele probleme un program de calcul face apel la alte programe specializate, cunoscute sub numele general de **subrutine** sau **subprograme**. În limbajul C subprogramele se cunosc sub numele de **funcții**.

Orice program scris in limbajul C se compune din funcții, dintre care una este funcția principală, numită **main**. Aceasta este lansată prima în execuție şi apelează alte funcții care, fie fac parte din biblioteca de funcții a limbajului, fie sunt create de utilizator. În exemplele prezentate pînă acum am creat numai funcția principala **main** şi am apelat funcțiile de bibliotecă pentru efectuarea operațiilor de citire/scriere (**printf**, **scanf**, **getche**, etc.) sau a operațiilor matematice (**sqrt**).

Utilizarea funcțiilor (subrutinelor) în elaborarea programelor de calcul vizează mai multe aspecte cu caracter general.

Un prim aspect îl constituie necesitatea de a nu fi nevoiți să scriem aceleaşi instrucțiuni de mai multe ori. Să presupunem că dorim să realizăm un program care să calculeze valoarea combinărilor prin formula: $C_n^k = n!/(k! \cdot (n - k)!)$. Acest program s-ar putea realiza, într-o maniera pe care o putem numi rudimentară, scriind de 3 ori setul de instrucțiuni pentru calculul factorialului în vederea evaluării termenilor *n!, k!* şi *(n-k)!*. Aceasta nu poate constitui o soluție, deoarece, dacă mai tîrziu în cadrul programului ar ap5rea necesitatea de a calcula valoarea C_n^k pentru alte valori ale lui *n* şi *k*, am fi nevoiți să repetăm setul de instrucțiuni. O manieră elegantă de rezolvare a problemei o constituie scrierea o singură dată a setului de instrucțiuni şi să se poată face apelul acestora în cadrul programului ori de cîte ori este necesar. Cu alte cuvinte, creăm o funcție care constă dintr-un set de operații specifice (de exemplu, calculul factorialului) care se efectuează asupra datelor de intrare (date globale ale programului de calcul şi/sau date transmise funcției prin intermediul parametrilor acesteia) şi asupra datelor proprii şi furnizează un set de rezultate (modificări efectuate asupra datelor globale, modificări efectuate prin intermediul parametrilor de tipul pointer, valoarea întoarsă de funcție sau alte acțiuni). În momentul apelului, funcția preia controlul programului, execută propriul set de instrucțiuni şi revine în funcția apelantă.

Un alt aspect de bază vizat de utilizarea funcțiilor îl constituie modularizarea programelor de calcul. În general, programele de calcul sunt realizate astfel încît să îndeplinească cît mai multe activități. Separarea şi plasarea acestora în cadrul unor funcții face munca de elaborare şi depănare a programului mult mai uşoară. De fapt, la realizarea programelor de calcul complexe se lucrează în echipă, fiecare programator realizînd una sau mai multe funcții specifice activităților programului.

Un al treilea aspect al folosirii funcțiilor îl constituie independența variabilelor utilizate de acestea. Cu alte cuvinte, variabilele declarate în cadrul unei funcții, numite şi **variabile locale**, sunt private, adică nu pot fi accesate de funcția principală sau de alte funcții. În acest fel este posibilă utilizarea aceluiaşi nume de variabilă în funcții diferite, ceea ce face mult mai uşoară elaborarea programelor complexe de mari dimensiuni.

6.2. Definirea şi declararea funcțiilor

Intrările într-o funcție şi rezultatele furnizate de aceasta constituie interfața funcției cu funcția apelantă. Utilizatorul funcției trebuie să înțeleagă numai această interfață, fără a fi preocupat de detaliile de realizare efectivă a funcției. În schimb, programatorul care proiectează şi implementează o funcție este preocupat atît de realizarea interfeței cît şi de modul de implementare efectivă a operațiilor pe care funcția le va efectua în momentul apelului.

Ideea de baza în realizarea unei funcții o constituie evitarea pe cît posibil a folosirii în datele de intrare în funcție a datelor globale specifice unui anume program. Prin aceasta, funcția capătă un caracter general şi poate fi considerată o operație abstractă ce se efectuează asupra parametrilor şi furnizează rezultate, putînd fi folosită în cadrul oricărui program de calcul. În aceste condiții, procesul de creare a unei funcții, cunoscut sub numele de **definirea funcției**, constă din:

- **stabilirea tipului funcției**, adică a tipului valorii pe care funcția o întoarce. Tipul funcției poate fi oricare dintre tipurile fundamentale de date: caracter cu sau fără semn, întreg simplu sau lung (cu sau fără semn), real, simplă sau dublă precizie, respectiv void, în cazul în care funcția nu întoarce nici o valoare.
- **stabilirea numelui funcției**. Pentru numele funcției se poate utiliza orice combinație de caractere (litere,cifre,_) cu condiția că primul caracter să fie o literă. De obicei, numele funcției este un cuvînt semnificativ pentru rolul pe care aceasta îl are de îndeplinit şi este separat de tipul acesteia printr-un spațiu.
- **stabilirea parametrilor**, adică stabilirea numărului şi a tipului parametrilor cu care funcția va fi apelată.
- **scrierea setului de instrucțiuni ce alcătuiesc corpul funcției** în conformitate cu rolul pe care aceasta îl are de îndeplinit. Corpul funcției este delimitat de o pereche de acolade şi cuprinde instrucțiunile de declarare a variabilelor locale şi instrucțiunile propriu-zise de prelucrare.

În concluzie, structura generală a unei funcții este:

```
tip nume_functie (lista parametrilor)
 {
declaratii de variabile locale
instructiuni simple sau compuse
 }
```

Prima linie din această construcție sintactică, ce conține tipul, numele funcției şi perechea parantezelor rotunde între care se află lista parametrilor, nu se termină cu caracterul ";". Pentru exemplificare, scriem programul pentru calculul combinărilor folosind o funcție pe care o numim factorial.

Exemplul 6.1.

```
/* Programul ex_6_1 */
#include <stdio.h>
unsigned long factorial(int); // declararea funcției
void main(void) {     /* Funcția principala */
        int n,k,cnk;
        clrscr( );
        printf("\nIntroduceti valorile lui n si k (n>=k)");  scanf("%d %d",&n,&k);
```

```
        cnk = factorial(n)/(factorial(k)*factorial(n-k));
        printf("\nValoarea combinarilor de %d luate cite %d este %d",n,k,cnk);    getch ();
}
unsigned long factorial(int m)   // Funcția pentru calculul factorialului
{
    int k; unsigned long fact = l;
    for(k=l; k<=m; k++)
        fact*=k;    return (f act);
}
```

Deoarece valoarea factorialului este un întreg pozitiv care în mod uzual depăşeşte capacitatea de memorare a unui întreg simplu, funcția factorial este definită de tipul întreg lung fără semn (unsigned long). Funcția are un singur parametru de tipul întreg *m*. Această maniera de definire a parametrilor prin indicarea tipului şi a numelui chiar în interiorul perechii de paranteze rotunde este în concordanță cu ultimele modificări aduse standardului limbajului de programare C. În maniera clasică, recunoscută de compilator, între parantezele rotunde se indică numai numele parametrilor, tipul acestora fiind precizat prin instrucțiuni separate plasate înaintea acoladei ce deschide corpul funcției. Deci, o scriere de forma:

```
        unsigned long factorial(m)
        int m;
```

este corectă, dar prezintă dezavantajul că este mai lungă.

În momentul apelului, variabila *m*, numită **parametru formal**, primeşte valoarea variabilei cu care se face apelul, numită **parametru actual**, care este diferită de la un apel la altul. În exemplul prezentat, mai întîi *m* va primi valoarea lui *n*, apoi pe cea a lui *k* şi în final valoarea expresiei *n-k*. Acest mecanism de transmitere a parametrilor este cunoscut sub numele de "**transmiterea parametrilor prin valoare**". Deoarece transmiterea parametrilor actuali se face prin valoare, este posibilă folosirea în apelul funcțiilor a constantelor, a variabilelor, a expresiilor sau chiar a numelor de funcții.

Revenind la funcția **factorial**, se constată că aceasta foloseşte pentru controlul buclei **for** variabila de tipul întreg *k*, folosită şi în cadrul funcției *main*. Acest lucru este posibil deoarece variabila *k*, declarată în interiorul funcției **factorial**, este o variabilă locală şi nu este cunoscută de funcția apelantă **main**.

O variabilă locală utilizată în corpul unei funcții este cunoscută în limbajul C sub numele de variabilă "**automatic**", deoarece ea este creată în mod automat în momentul apelului funcției şi distrusă în momentul revenirii în programul apelant. Intervalul de timp în care variabila locală este utilizată poartí numele de "**dutată de viață**" (lifetime).

După calculul factorialului în cadrul buclei **for**, funcția revine în programul principal (funcția **main**) furnizîndu-i acestuia valoarea calculată, prin intermediul instrucțiunii **return**. Referitor la această instrucțiune se fac următoarele precizări:

- întoarce funcției care face apelul o singură valoare de tipul asociat la definirea funcției.
- se pot utiliza mai multe instrucțiuni **return** pentru a reveni în funcția care face apelul din diverse puncte ale funcției apelate, ca urmare a execuției unor instrucțiuni de decizie în corpul acesteia.

De exemplu, dacă dorim ca funcția **factorial** să întoarcă valoarea 1, în cazul în care valoarea parametrului actual este mai mică sau egală cu 1, fără a executa bucla, atunci setul de instrucțiuni ce definesc corpul acesteia se rescrie astfel:

```
        ...
        if (m<=1) return(1);
        else {
            for (k=1; k<=m; k++) fact *= k;
            return(fact);
        }
```

Revenind la programul din exemplul 6.1 constatăm că pe lîngă directiva *#include <stdio.h>*, care atenționează compilatorul că se vor folosi funcțiile standard de intrare/ieşire, s-a folosit instrucțiunea *unsigned long factorial(int);*. Această instrucțiune este un exemplu de declararea unei funcții şi are rolul de a semnala compilatorului că în cadrul programului se va folosi o funcție avînd tipul şi numele precizat înaintea perechii de paranteze rotunde, iar tipul fiecărui parametru formal este precizat în interiorul acestora. Menționăm că se precizează numai tipul parametrilor formali, nu şi numele acestora şI că instrucțiunea de declarare se termină cu ";".

Declararea funcțiilor este o facilitate importantă oferită de limbajul C, deoarece în faza de compilare se va verifica la fiecare apel dacă numărul şi tipul parametrilor actuali corespund cu cei ai parametrilor formali utilizați la definirea funcției, adică dacă apelul este făcut corect.

6.3. Utilizarea mai multor funcȚii în program

Pentru realizarea unui program de calculeste necesar să definim şi să utilizăm mai multe funcții. Acest lucru este permis în limbajul C, dar comparativ cu limbajul Pascal există 2 diferențe fundamentale. Prima constă în faptul că în C nu este permisă definirea unei funcții în interiorul altei funcții, lucru permis în Pascal.Deci, în limbajul C toate funcțiile sunt definite în mod independent. A doua deosebire constă în faptul că în Pascal o funcție (procedură) definită în cadrul altei funcții nu poate fi apelată de o funcție din exterior, în timp ce în C o funcție poate apela toate celelalte funcții.

Vom exemplifica modul de comunicare între funcțiile limbsjului C, considerînd programul din exemplul 6.2, care utilizează 3 funcții pentru a cacula suma pătratelor a 2 numere reale introduse de la tastieră.

Exemplul 6.2.

```
/* Programul ex_6_2 */
#include <stdio.h>
float suma(float,float);    float patrat(float);      float suma_patrate(float,float);  // declararea funcțiilor
void main(void) {     /* funcția principala */
    float a,b; clrscr( );
    printf("\nIntroduceti 2 valori reale > ");     scanf("%f %f",&a,&b);
    printf("\n Suma patratelor este %f",suma_patrate(a,b));   getch ();
}
```

```
/* definirea funcţiilor */
float patrat(float x) {
        return(x*x);
}
float suma(float xl, float x2) {
        return(xl+x2);
}
float suma_patrate(float y1,float y2) {
        return(suma(patrat(yl),patrat(y2)));
}
```

Toate cele 3 funcţii furnizează o valoare reală şi deci au tipul float. Prima funcţie **patrat** are un singur parametru *x* de tipul real, şi returnează funcţiei care o apelează valoarea pătratului acestuia. A doua funcţie **suma** are doi parametri de tipul real *x1* şi *x2* şi returnează funcţiei care o apelează valoarea sumei acestora. Ultima funcţie **suma_patrate** are doi parametri de tipul real *y1* şi *y2* şi returnează funcţiei care o apelează valoarea sumei pătratelor acestora.

Pentru afişarea sumei pătratelor valorilor *a* şi *b* introduse de la tastatură, funcţia **main** apelează funcţia **printf**, care la rîndul ei apelează funcţia **suma_patrate** cu parametrii actuali *a* şi *b*. Funcţia **suma_patrate**, pentru a evalua expresia *a+b*, apelează funcţia **suma**, care are ca parametri actuali două apeluri la funcţia patrat. Astfel, valorile variabilelor *a* şi *b* sunt transferate pe rînd funcţiei **patrat** care furnizează valorile a^2 şi b^2. Acestea sunt transmise apoi funcţiei **suma**, care furnizează valoarea a^2+b^2, iar în final aceasta este transmisă de către funcţia **suma_patrate** funcţiei **printf** care afişează rezultatul pe ecran.Desigur, acest lucru se putea realiza foarte simplu utilizînd în apelul funcţiei **printf** expresia a*a+b*b. Dar programul a fost complicat în mod voit pentru a pune în evidenţă modul în care pot fi utilizate funcţiile în limbajul C.

Aşadar în limbajul C funcţiile sunt tratate ca şi variabilele. Ele pot fi folosite în cadrul expresiilor aritmetice sau logice, în cadrul apelului altor funcţii pe post de parametri actuali sau în cadrul instrucţiunilor **return**. Pentru a evidenţia acest lucru, vom rescrie funcţia **suma_patrate**, utilizînd mai multe variabile locale.

```
float suma_patrate(float yl, float y2) {
        float patrat1,patrat2,sum;
        patratl = patrat(yl);    patrat2 = patrat(y2);
        sum = suma(patratl,patrat2);      return(sum);
}
```

Din punct de vedere funcţional, această versiune a funcţiei **suma_patrate** este identică cu cea folosită în exemplul anterior. Este mult mai clară, dar prezintă dezavantajul că foloseşte trei variabile suplimentare (*patrat1*, *patrat2* şi *sum*) pentru a memora valorile *y1*, *y2* şi, respectiv, *y1+y2*. Utilizarea acestora este inutilă deoarece, aşa cum s-a putut constata, în locul lor se pot folosi funcţiile care furnizează valorile respective.

În concluzie, pentru a realiza programe performante în limbajul C se vor utiliza, ori de cîte ori este posibil, apeluri directe de funcţii, fără folosirea unor variabile intermediare, care măresc dimensiunea programului.

De menţionat, că ordinea în care sunt declarate şi definite funcţiile într-un program scris în limbajul C este la altitudinea programatorului. De exemplu, acesta poate să-şi declare şi să-şi definească funcţiile în ordine alfabetică sau pur şi simplu aleator.

6.4. Funcţii de tipul void

Din cele prezentate pînă acum rezultă că orice funcţie are un tip care defineşte tipul valorii pe care aceasta o întoarce. În practică, există funcţii care nu întorc o valoare. Considerăm pentru exemplificare programul din exemplul 6.3.

Exemplul 6.3.

```
/* Programul ex_6_3 */
                void dreptu(int,int);
                void eroare(void);
                void main (void)
                 {
                int lung, lat;
                printf ("\nIntroduceti lungimea si latimea dreptunghiului >");    scanf ("%d %d",&lung,&lat);
                dreptu(lung,1at);
                getch( );
                 }
                void dreptu(int i,int j)
                 {
                int m,n;
                if (i<=0 || i>=80 || j<=0 || j>=25)
                    eroare ();
                else
                 {
                    clrscr( );
                    for (m=l; m<=j; m++)
                     {
                        for (n=l; n<=i; n++)
                            printf ("\xB0");
                        printf ("\n");
                     }
                 }
                 }
```

```
void eroare( )
 {
int i;
printf (“\x7”);
for(i=0; i<=20000; i++)
    ;    /* instructiune vida */
printf (“\x7”);
printf (“\nDimensiuni eronate. Apasati o tasta”);  getch( );
 }
```

În cadrul programului este utilizată funcția **dreptu** pentru trasarea pe ecran a unui dreptunghi folosind un caracter grafic corespunzător secven.ei escape \xDB. Această făncție are ca parametri formali 2 variabile de tipul întreg care semnifică lungimea, respectiv lățimea dreptunghiului şi nu returnează o valoare. Pentru a defini şi declara o funcție care nu întoarce o valoare, în limbajul C se foloseşte cuvîntul cheie **void** ca tip de funcție.

În vederea trasării dreptunghiului, funcția **dreptu** mai întîi verifică dacă dimensiunile acestuia se încadrează în dimensiunile ecranului, care sunt în mod uzual 80 de coloane şi 25 de linii, folosind în cadrul instructiunii **if..else** expresia logică:

```
i<=0 || i>=80 || j<=0 || j>=25
```

Dacă se constată o depăşire a dimensiunilor permise, se apelează funcția **eroare** care afişează pe ecran mesajul: “*Dimensiuni eronate. Apăsați o tastă*” precedat de doua semnale sonore.

Funcția **eroare** este un caz aparte de funcție, în sensul că nu are parametri formali (argumente) şi nici nu întoarce o valoare. Pentru a declara o astfel de funcție se foloseşte cuvîntul cheie **void** atît pentru definirea tipului funcției, cît şi pentru definirea tipului argumentului. La definire, în locul parametrilor formali se foloseşte un spațiu între cele două paranteze rotunde.

Referitor la corpul funcției **eroare**, remarcăm utilizarea secventei escape \x7 în cadrul funcției **printf**, care are ca efect emiterea unui semnal sonor, precum şi a buclei **for** urmată de caracterul “;”.După emiterea primului semnal sonor se execută bucla for care conține o instrucțiune vidă (caracterul “;”) pentru valori ale lui *i* de la 0 la 20000. Această manieră de programare a buclei for este folosită pentru a crea o pauză între execuția a 2 instrucțiuni succesive (în cazul de față, emiterea celor 2 semnale sonore). Durata pauzei este controlata prin valoarea maximă pe care trebuie să o atinga variabila de control a buclei, adica prin expresia de decizie a ieşirii din buclă. O altă posibilitate de creare a unei pauze o constituie apelul funcției **delay** definita în fişierul antet <**dos.h**>. Forma generală de apel a acestei funcții este: *delay(pauza);* în care parametrul *pauza* poate fi o constantă sau o variabilă de tipul întreg a cărei valoare specifică durata pauzei, exprimată în milisecunde.

6.5. Variabile globale

În exemplele prezentate pînă acum, declararea variabilelor s-a făcut fie în interiorul funcției principale **main**, fie în interiorul celorlalte funcții. Limbajul C oferă posibilitatea declarării variabilelor din cadrul unui program în afara oricărei funcții. O variabilă declarată în acest mod poartă numele de **variabilă globală** şi are proprietatea că poate fi accesată de toate funcțiile care sunt definite după instrucțiunea de declarare a acesteia. Considerăm pentru exemplificare programul din exemplul 6.4.

Exemplul 6.4.

```
/* Programul ex_6_4 */
void citire(void);
void impartire(void);
void rezultat(void);
/* declararea coastantelor si variabilelor globale */
const float infinit=3.4e+38;
float a,b,cit;
void main(void)
 {
citire( );   impartire( );   rezultat( );getch( );
 }
void citire( )
 {
clrscr( );   printf (“Introduceti 2 valori reale >”);  scanf(“%f %f”,&a,&b);
 }
void impartire( )
 {
if (b==0)  cit=infinit;
else       cit=a/b;
 }
void rezultat( )
 {
if (cit==infinit)   printf (“\nOperatie fara sens”);
else                printf (“\na=%f b=%f a:b=%f”,a,b,cit);
 }
```

Programul citeşte de la tastatură 2 valori reale, efectuează împărțirea acestora şi afişează rezultatul. Pentru aceasta sunt folosite 3 funcții: **citire**, **cit** şi **rezultat**, definite de tipul **void** şi fără parametri. Transmiterea valorilor de la o funcție la alta se fa-ce prin intermediul variabileior globale *a, b, cit* şi *infinit* declarate la începutul programului şi care sunt accesibile tuturor func-țiilor. *infinit* a fost definită ca constantă, avînd valoarea 3.4e+38 (cea mai mare valoare reală ce poate fi memorată într-un real).

Spre deosebire de variabilele locale, care se distrug în momentul părăsirii funcției, variabilele globale rămîn în memorie pe toată durata execuției programului, adică au durata de viață egală cu cea a programului.

Variabilele globale pot fi declarate oriunde în interiorul unui program, dar funcţiile definite înaintea lor nu le vor recunoaşte. Astfel, dacă în programul anterior vom muta declaraţiile variabilelor globale după definirea funcţiei **citire** şi înainte de funcţia **cit**, compilatorul va semnala 2 erori generate de faptul că variabilele *a* şi *b* nu sunt definite. Deci, ca orice variabilă, şi variabilele globale trebuie definite înainte de a fi utilizate.

6.6. Directive de preprocesare

Directivele de preprocesare constituie comenzi adresate compilatorului, pe care acesta trebuie să le execute înainte de efectuarea compilării propriu-zise. Se poate vorbi deci de un limbaj în interiorul limbajului C. Această facilitate, specifică limbajelor de asamblare, ne permite să afirmăm că limbajul C se apropie de limbajul maşinii.

Pentru a înţelege mai bine directivele de preprocesare, să analizăm mai întîi rolul compilatorului. Cînd scriem o instrucţiune de forma: *numar = 56.32;* cerem compilatorului să o transforme într-un set de instrucţiuni care să poată fi executate de microprocesor. Spre deosebire de aceste instrucţiuni, directivele de preprocesare se adresează compilatorului cerîndu-i mai întîi să modifice textul programului (sursa) şi apoi să efectueze compilarea. De aici provine şi numele de **preprocesare**.

Pentru a specifica o directivă de preprocesare se foloseşte semnul **#** în faţa acesteia.Vom analiza în continuare 2 dintre cele mai utilizate directive de preprocesare, directiva **#define** şi directiva **#include**.

Cea mai simplă utilizare a directivei **#define** este de a atribui un nume unei constante. Pentru exemplificare, considerăm programul din exemplul 6.5, în care se utilizează directiva **#define** pentru a defini constanta matematică *n*, necesară calculării ariei şi volumului unei sfere.

Exemplul 6.5.

```
/* Programul ex_6_5 */
#include <stdio.h>
#define    PI  3.14159
double arie(float);
double volum(float);
void rnain(void)
 {
float raza;
clrscr( );
printf (“Introduceti raza sferei > ”);scanf (“%f ”,&raza);
printf (“\nAria sferei de raza R=%g este S=%g, iar volumul V=%g”,raza,arie(raza),volum(raza));   getch( );
 }
double arie(float r)
 {
return (4*PI*r*r);
 }
double volum(float r)
 {
return(4*PI*r*r*r/3);
 }
```

Programul solicită introducerea unei valori reale care reprezintă raza sferei şi afişează aria şi volumul acesteia. În acest sens sunt utilizate funcţiile **arie** şi **volum**. Rolul directivei *#define PI 3.14159*, care începe cu semnul # şi nu se termină cu caracterul “;”, este de a atribui constantei 3.14159 numele simbolic PI. Cînd compilatorul întîlneşte această directivă, mai întîi va modifica textul programului substituind cuvîntul PI în toate expresiile sau instrucţiunile în care apare cu valoarea numerică 3.14159 şi apoi va efectua compilarea.

Directivele **#define** pot fi plasate oriunde în corpul programului, dar este de preferat ca ele să fie la început. În ceea ce priveşte numele atribuit unei constante, acesta se scrie cu litere mari. Avantajul utilizării directivei **#define** constă în faptul că atunci cînd dorim să modificăm valoarea constantei vom face acest lucru numai în directivă şi vom recompila programul. De exemplu, dacă dorim ca valoarea lui *n* să fie introdusă cu şase zecimale, este suficient să modificăm directiva astfel: *#define PI = 3.141592*. Recompilînd programul, în faza de preprocesare modificarea făcută se va propaga în tot fişierul, astfel că noua valoare se va substitui cuvîntului PI în toate poziţiile în care acesta apare.

Pe lîngă faptul că permite definirea şi modificarea rapidă a valorilor unei constante, directiva **#define** oferă o gamă largă de facilităţi care provin din faptul că poate utiliza argumente. Pentru exemplificare, considerăm următorul program de calcul:

Exemplul 6.6.

```
/* Programul ex_6_6 */
#include <stdio.h>
#define    PI  3.141592
#define    ARIE(x)   4*PI*x*x
#define    VOLUM(y)   4*PI*y*y*y/3
void main(void)
 {
float raza,s,v;
clrscr( );
printf (“Introduceti raza sferei > ”);scanf(“%f”,&raza);
s=ARIE(raza);    v=VOLUM(raza);
printf(“\nAria sferei de raza R=%g este S=%g, iar volumul V=%g”,raza,s,v);  getch ();
 }
```

Programul utiiizează directiva **#define** pentru a defini formulele de calcul ale suprafeței şi volumului unei sfere. În faza de precompilare, instrucțiunile: *s=ARIE(raza); v = VOLUM(raza)* vor fi transformate în instrucțiunile: *s=4*3.141592*raza*raza;* respectiv *v=4*3.141592*raza*raza*raza/3;*. Mai întîi este substituită valoarea lui PI ca efect al primei directive **#define**, după care, în momentul apelului, parametrii *x* şi *y* sunt substituiți în formulă cu cuvîntul *raza*, rezultînd instrucțiunile care vor fi folosite în faza de compilare.

Utilizarea directivei **#define** în această formă poartă numele de "**macro**". Din exemplul prezentat, constatăm că putem utiliza "macro"-uri în locul funcțiilor. Prin utilizarea "macro"-urilor, datorită mecanismului de substituire din faza de precompilare, aceeaşi secvență de instrucțiuni va fi repetată de mai multe ori în cadrul aceluiaşi program. Deci, codul generat este mai mare, dar programul devine mai rapid. Utilizarea funcțiilor determină evitarea repetării aceleiaşi secvențe de instrucțiuni, dar are dezavantajul că, datorită mecanismului de transmitere a parametrilor, viteza de execuție scade. În concluzie, decizia alegerii între "macro"-uri şi funcții depinde de compromisul ce trebuie realizat între viteza de calcul şi economia de memorie.

Un aspect esențial care trebuie avut în vedere la utilizarea "macro"-urilor îl constituie modul în care sunt construite expresiile din directiva **#define**, precum şi modul în care sunt apelate. Să considerăm, de exemplu, "macro"-ul care execută suma a două numere;

```
#define    SUMA(x,y)   x+y
```

pe care-l utilizăm într-o expresie de forma:

```
val = 10+SUMA(3,2);
```

în scopul obținerii valorii 50. Vom constata că, în urma execuției instrucțiunii, valoarea variabilei *val* este 32, deoarece prin substituirea "macro"-ului instrucțiunea devine:

```
val = 10*3+2;.
```

Avînd în vedere precedența operatorilor aritmetici, rezultă că valoarea corectă este cea afişată, adică 32. Pentru a obține valoarea dorită, 50, se vor utiliza parantezele, fie în definirea "macro"-ului care se poate scrie astfel:

```
#define SUMA(x,y) (x+y)
```

fie în apelul acestuia, scriind instrucțiunea de apel astfel:

```
val = 10*(SUMA(3,2));
```

0 altă directivă de preprocesare este **#include** şi are rolul de a solicita compilatorului că, în faza de preprocesare, acesta să includă - în fişierul sursă în care este folosită - un alt fişier, cu scopul de a fi compilate împreună. În mod curent, directiva **#include** este folosită pentru includerea fişierelor antet.

Mediile de programare Turbo C, Turbo C++ şi Borland C au biblioteci de funcții care conțin, pe lîngă declarațiile funcțiilor, numeroase "macro"-uri. Acestea sunt definite în fişiere numite **fişiere antet** sau **header**-e. Numele acestor fişiere sunt formate dintr-un cuvînt ce specifică rolul funcțiilor definite în ele, iar extensia este "h". Astfel, funcțiile de intrare/ieşire sunt definite în **stdio.h**, cele matematice în **math.h**, cele grafice în **graphics.h**.

Pentru a solicita includerea unui fişier antet al mediului de programare, se foloseşte directiva **#include** urmată de numele fişierului cuprins între paranteze unghiulare. Deci, directiva *#include <stdio.h>* utilizată în toate exemplele prezentate pînă acum are ca efect includerea în fişierul sursă a fişierului **stdio.h**.

Utilizatorul poate să-şi creeze propriile fişiere antet, pe care să le folosească în cadrul diverselor programe, cu ajutorul directivei **#include**. Pentru exemplificare, considerăm că realizăm un fişier antet, numit **volume.h**, care conține următoarele "macro"-uri pentru calculul volumelor diverselor corpuri geometrice:

```
#define PI      3.141592
#define VOL_SFERA(rs)    4*PI*rs*rs*rs/3
#define VOL_CON(rc,hc)   PI*rc*rc*rc/3
#define VOL_CIL(rcil,hcil) PI*rcil*rcil*hcil
#define V_CUB(l)             l*l*l
#define V_PARAL(lung,lat,inalt) lung*lat*inalt
```

Fişierul este realizat folosind editorul de texte şi salvat în catalogul curent de lucru. Utilizarea lui în cadrul unui program se face cu directiva *#include "volum.h"* în care numele fişierului este cuprins între ghilimele.

Deci, dacă dorim includerea unui fişier antet al mediului de programare, folosim numele acestuia cuprins între parantezele unghiulare < >, iar dacă dorim includerea unui fişier din catalogul de lucru curent, folosim numele fişierului, cuprins între ghilimele.

Pentru exemplificare, considerăm programul din exemplul 6.7, care foloseşte fişierul antet utilizator **volume.h**, creat anterior, şi fişierul antet al mediului de programare, **stdio.h**.

Exemplul 6.7.

```
/* Programul ex_6_7 */
#include<stdio.h>
#include "volume.h"
int meniu(void);
void main(void)
 {
int caz;
float a,b,c;
while ((caz=meniu( )) != 0)
 {
    switch (caz)
     {
    case 1:
        clrscr ( );
        printf("Introduceti raza sferei > ");          scanf("%",&a);
        printf("\nVolumul sferei Vs=%g",V_SFERA(a));
        getch( );       break;
```

```
        case 2:
            clrscr( );
            printf("Introduceti raza si inaltimea conului >"); scanf("%f %f",&a,&b);
            printf("\nVolumul conului Vcon=%g",V_CON(a,b));
            getch( );   break;
        case 3:
            clrscr( );
            printf("Introduceti raza si inaltimea cilindrului >");   scanf("%f %f",&a,&b);
            printf("\nVolumul cilindrului Vcil=%g",V_CIL(a,b));
            getch( );   break;
        case 4:
            clrscr( );
            printf("Introduceti latura cubului >"); scanf("%f",&a);
            printf("\nVolumul cubului Vcub=%g",V_CUB(a));
            getch( );   break;
        case 5:
            clrscr( );
            printf("Introduceti laturile paralelipipedului >"); scanf("%f %f %f ",&a,&b,&c);
            printf("\nVolumul paralelipipedului Vpar=%g",V_PARAL(a,b,c));
            getch( );   break;
       }
    }
    }
    int meniu( )
     {
    int nr;
    clrscr( );
    gotoxy(30,5); printf("CALCULUL VOLUMELOR");
    gotoxy (35,7);     printf("1 sfera");
    gotoxy (35,8);     printf("2 Con");
    gotoxy (35,9);     printf("3 Cilindru");
    gotoxy (35,10);    printf("4 Cub");
    gotoxy (35,11);    printf("5 paralelipiped");
    gotoxy (35,12);    printf("0 stop");
    gotoxy (35,14);    printf("selectati o varianta >");       scanf(" %d",&nr);
    return(nr);
     }
```

Programul calculează volumul unui corp geometric în funcție de varianta aleasă din meniul afişat.

6.7. Apeluri recursive

Spunem despre o funcție că este recursivă dacă ea se autoaplică. Ea se poate autoapleca fie direct, fie indirect prin apelul altor funcții. În C++ nu se spicifică.

La apelurile recursive ale unei funcții aceasta este reapelată înainte de a se reveni la ea. De aceea, aceste clase au valori distincte la fiecare reapelare. Variabilele statice ocupă tot timpul aceeaşi zonă de memorie şi deci ele îşi păstrează valoarea la un reapel.

Orice apelal unei funcții conduce la o revenire din funcția respectivă la punctul următor celui din care s-a făcut apelul.

La revenire dintr-o funcție se procedează la curățirea stivei, adica stiva se reface la starea ei dinaintea apelului. De aceea orice reapel al unui apel recursiv va conduce şi el la curățirea stivei, la o revenire din funcție şideci parametrii şi variabelele locale vor reveni la valorile lor dinainte reapelului respectiv. Numai variabilele statice rămîn neschimbate la o astfel de revenire.

Funcțile recursive se utelizează la defenirea procesilor recursive de calcul.Un proces de calcol se spune că este recursiv ,dacă el are o parte care se defineşte prin el însuşi.Un proces recursiv trebuie totbeauna să conțină o parte care nu se defineşte prin el însuşi .

Să definim ca recursiv procesul de calcol al factorialului:

```
double fact (int n) {
    if (n==0) return 1.0;
    else return n*fact(n-1);
}
```

În general, ofuncție recursivă se poate realiza şi nerecursiv, adica fără să se auroapelare.De obicei, recursivitatea nu conduce la economie de memorie şi nici la execuția mai rapidă a programelor. Ea permite însă o descriere mai complicată şi mai clară a funcțiilor care exprimă procese de calcol recursive. Cutoate acestea, există cazuri cînd funcțiile recursive, deşi au exprimări clare, ele nu sînt recomandate deoarece implică consum mare de timp şi memorie,în comparație.