Funcții

1. Definirea şi apelul funcțiilor

Funcțiile sunt elementele constructive ale limbajului C si sunt de două categorii:

- funcții standard (de bibliotecă) definite în fisiere *.h: **`printf(), scanf(), sin(), strlen(), getchar(), putchar(), . . .`** ; acestea pot fi utilizate în programe, prin specificarea fisierului header în care se află.
- funcții definite de utilizator, altele decât cele standard.

În programul sursă, o funcție definită de utilizator, are două părți: *antetul* si *corpul funcției*. O funcție poate avea un număr de parametri (variabile) dar o singură valoare - *valoarea funcției*. Antetul conține informații despre tipul valorii funcției, numărul si tipul parametrilor si conține identificatorul (numele) funcției.

Structura unei funcții:

<tip> <nume> (<lista declarați ilor parametrilor formali>)
{
<declarații>
<instrucțiuni>
}

Primul rând este antetul; acesta apare obligatoriu înaite de corpul funcției dar si la începutul programului (înainte de main()) situație în care se numeste *prototip*.

În cazul tipurilor standard, <tip> din antet este un cuvânt cheie care defineste tipul valorii returnate de funcție.

În C există două categorii de funcții:

- funcții cu tip, care returnează o valoare, de tipul <tip>;
- funcții fără tip, care nu returnează nici o valoare după execuție, dar efectuează anumite prelucrări de date; pentru acestea se va folosi cuvântul rezervat ***void*** în calitate de <tip>.

O funcț ie poate avea zero, unul sau mai mulți parametri. Lista declarațiilor parametrilor (formali) este vidă când funcția nu are parametri.

În corpul funcției pot să apară una sau mai multe instrucț iuni return, care au ca efect revenirea în programul apelant. Dacă nu apare nici o instrucțiune return, revenirea în programul apelant se face după execuția ultimei instrucț iuni din corpul funcției.

Valoarea returnată de o funcție. Instrucțiunea return

Forma generală: return <expresie> ;

unde "expresie" trebuie să aibă o valoare compatibilă cu tipul declarat al funcț iei. Efectul constă în încheierea execuției funcției si revenirea în programul apelant; se transmite valoarea calculată <expresie> către programul apelant, reprezentând **valoarea funcției**, ce va înlocui **numele funcției**. Dacă funcț ia este de tip void, expresia lipseste.

Exemplu:

```
void fact(int n); //calculeaza n!
{
int k; long fct;
if (n<0) {
printf("Valoarea arg. negativa !"\n);
return; //nu se calculeaza n!
}
for (fct=1, k=1; k<=n; k++)
fct *=k;
printf(Valoarea %d!=%ld\n", n, fct);
return; // s-a calculat n! si s-a afisat
}
```

- Dacă într-o funcție sunt mai multe comenzi return, prima ce se execută va produce revenirea în programul apelant.

- O funcție poate transmite prin return o singură valoare, de orice tip (inclusiv structură) dar fără tipul tablou.

- Valoarea returnată de o funcție cu tip trebuie să fie compatibilă cu tipul declarat al funcției; de exemplu, dacă o funcție este definită prin int f(), valoarea returnată poate fi de orice tip aritmetic, deoarece aceasta poate fi convertită imediat la tipul întreg:

```
int f()
{
int i, float a; char c;
. . . . . .
return i; //valoarea coincide cu tipul funcției
. . . . . .
return a; //valoarea "a" este convertita la int
. . . . . .
return c; //valoarea "c" este convertita la int
. . . . . .
}
```

- Dacă o funcție returnează o valoare de tip adresă (pointer), cerințele sunt mai restrictive: **tipul valorii** returnate trebuie să fie exact **tipul funcției** .

```
char *f()
{
char c, *adr_c, tablou_c[5], **adr_adr_c;
int *adr_i;
. . . . . .
c='A' ;
adr_c=&c;
return adr_c ; //corect, adr_c = tipul functiei
. . . . . .
*adr_adr_c=adr_c;
return *adr_adr_c;//corect,*adr_adr_c=tipul
functiei
adr_c=tablou_c;
return tablou_c; //corect, numele tabloului e un
//pointer de acelasi tip
```

```
//cu tipul functiei
. . . . . .
return *adr_c;//gresit, se returneaza un caracter
//in loc de adresa
. . . . . .
return adr_i; //gresit, tipul pointerului (int) nu
//corespunde cu tipul functiei
. . . . . .
return ; //valoarea returnata nu este definita
```

În cazul în care funcția are mai mulți parametri, declarațiile de tip pentru parametri se separă prin virgulă. Parametrii se utilizează pentru a permite transferul de date, către o funcț ie, în momentul utilizării ei în program. La construirea unei funcții, se face abstracție de valorile concrete. Acestea vor fi prezente numai la execuția programului.

Exemple:

void f(int a, int b, char c) /*corect, se specifica tipul fiecarui param.*/

void f(int a, b, char c) /* incorect, nu se speci fica tipul lui b */

Parametrii unei funcții, ca si variabilele definite în corpul funcției sunt *variabile locale*, adică sunt valabile numai în interiorul funcției; ele nu sunt recunoscute în exterior, nici ca nume, nici ca valoare.

În momentul compilării, este necesară doar cunoasterea tipurilor valorilor pe care le vor primi parametrii la execuție. Aceste declarații de tip sunt indicate în antetul funcției si vor servi compilatorului să rezerve memoria necesară pentru fiecare parametru.

Parametrii declarați în antetul unei funcții se numesc *formali,*
pentru a evidenția faptul că ei nu reprezintă valori concrete, ci numai țin locul acestora, pentru a putea exprima procesul de calcul. Ei se concretizează la execuție prin valori ce rezultă din apelarea funcț iei, când sunt numiți *parametri efectivi* .

Observații:

- Dacă nu se specifică tipul unei funcții, se consideră automat că ea va returna o valoare de tip *int* .

- În limbajul C++, controlul cu privire la tipul valorii unei funcții este mai strict si ca urmare se recomandă, ca tipul să fie efectiv specificat, în toate cazurile.

- Pentru funcția principală ***main***, se pot utiliza antetele:

int main()
int main(void)
void main()
void main(void)
main()
main(void)

Primele două antete arată că funcția returnează o valoare întreagă; de asemenea ultimele două antete arată că se returnează o valoare întreagă, chiar dacă nu se specifică tipul *int* în antet.

Se utilizează foarte frecvent varianta fără tip main() .

Funcția *main* poate avea si parametri.

Programul următor produce tabelarea valorilor unei funcții de două variabile, în domeniul [0, 1]x[0, 1], cu pasul 0.1.

```
# include <stdio.h>
double fm(double, double);
void main(void)
{
double x, y, pas=0.1;
for(x=0.0; x<=1.0; x=x+pas)
for(y=0.0; y<=1.0; y=y+pas)
printf("x=%lf y=%lf f(x,y)=%lf\n",x,y,fm(x,y));
{
double fm(double x, double y)
{
return (3.0*x*y + 1.0)/(1.0 + x + y + x*y);
}
```

Programul conține două funcții: main() si fm(). Funcția matematică fm() are doi parametri de tip real, dublă precizie si anume x si y iar tipul valorii funcției este tot real, dublă precizie. Numele funcției reprezintă valoarea întoarsă de aceasta, de exemplu, fm(0.0, 1.0)=0.5 este valoarea lui fm când argumentele sale iau valorile x=0 si y=1.

Instrucțiunea return <expresie>, întoarce valoarea expresiei în programul apelant, această formă fiind obligatorie la funcțiile cu tip.

Declarația

double fm(double, double) ;

de la începutul programului este un *prototip* al funcției fm(), care descrie tipul funcției, numele si tipul fiecărui parametru. În acest mod, funcția fm() este recunoscută în main(), chiar dacă definiția ei se află după main() .

În general, este indicat să se scrie prototipurile tuturor funcțiilor, atât ale celor definite în modulul curent de program, cât si ale celor definite în alte module dar folosite în modulul curent.

Funcțiile de bibliotecă sunt complet definite (inclusiv prototipurile) în fisiere header, deci includerea acestor fisiere, asigură si prezența prototipurilor. De exemplu, prototipul lui printf() este în fisierul stdio.h.

Apelul funcțiilor se face diferențiat, după cum sunt cu tip sau fără tip. O funcție fără tip se apelează prin:

nume(*listă de parametri actuali*); /*cand funcția are parametri

nume(); /* cand functia nu are parametri

O funcție cu tip se apelează în general prin:

variabilă = nume(*listă de parametri actuali*);

sau, mai general, prin specificarea numelui funcției într-o expresie în care este permis tipul valorii funcției. Uneori nu ne interesează valoarea pe care o returnează o funcție si atunci funcția se apelează exact ca o funcție fără tip. Un exemplu uzual este funcția printf(), care întoarce numărul de caractere scrise la consolă, dar apelul uzual al acestei funcții se face ca si cum ar fi de tip *void*.

Funcțiile au clasa de alocare implicită external, deci sunt vizibile în toate modulele care compun aplicația. Ca si la variabilele externe, o funcție este vizibilă din locul unde este declarată, până la sfârsitul fisierului sursă. Din acest motiv se foloseste un prototip al funcției care se plasează la începutul textului sursă, în afara tuturor definițiilor

de funcții. O altă posibilitate este includerea prototipurilor în fisiere **header**. Fisierele header predefinite conțin, între altele, prototipuri ale funcțiilor de bibliotecă. Prezența prototipurilor face posibilă verificarea la compilare a corespondenței dintre numărul si tipul parametrilor formali cu numărul si tipul parametrilor actuali (la apelare). Exemple de prototipuri:

1. float fmat (float x, float y, int n)
2. int fdet (float, int, int, char)

Declarațiile parametrilor formali pot conține numele acestora (ca în exemplul 1) sau pot fi anonime, deci fără a specifica un nume pentru fiecare parametru (ca în exemplul 2).

În limbajul C, ordinea evaluării parametrilor actuali la apelul unei funcții, nu este garantată. În fapt, această ordine este în general de la dreapta la stânga, dar programele trebuie asfel construite încât să funcționeze corect indiferent de această ordine. De exemplu, secvența:

```
int n=7;
printf (" %d %d\n", n++, n);
```

va tipări (la Borland C): 7 7 si nu 7 8 cum ar fi de asteptat. Asemenea construcții trebuie evitate, prin separarea în mai multe apeluri, de exemplu:

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

Acelasi lucru este valabil si la expresii în care intervin apeluri de funcții. De exemplu, în secvența următoare nu se garantează care funcție va fi apelată prima:

```
y = f( ) - g( ) ;
```

Dacă f() este apelată prima, si calculul valorii lui f() are efecte asupra lui g() nu se obține acelasi rezultat ca în cazul apelării mai întâi a lui g(). Dacă ordinea este importantă, aceasta se poate impune prin:

```
y = -g( ) ; y = y + f( );
```

Exemplul 1 - (varianta 1, corectă)

Programul ilustrează necesitatea folosirii prototipului. Se calculează valoarea maximă dintr-un tablou de până la 20 de numere reale, introduse de utilizator.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
double max(double *tab, int n);//prototipul
functiei
void main()
{
double x[MAX];
int i;
printf("\n Tastati numerele,terminati cu 0:\n");
for(i=0; i<MAX; i++)
{
scanf("%lf",&x[i]);
if(x[i]==.0) break;
} // la iesire din for, i=nr.elemente introduse
```

```
printf("\n Maximul este %lf ", max(x,i));
getch();
}
double max(double *tab,int n)
/* functia max (),de tip double; parametrii sunt:
adresa tabloului si numarul de elemente */
{
int i; double vmax=*tab;
for(i=1;i<n;i++)
if(vmax<*(tab+i))
vmax=*(tab+i);
return vmax;
}
```

Exemplul 1 - (varianta 2, incorectă)

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
void main()
{
double x[MAX];
int i;
printf("\n Introd. numerele,terminati cu 0:\n");
for(i=0;i<MAX;i++)
{
scanf("%lf",&x[i]);
if(x[i]==.0) break;
} // la iesire din for, i = nr.elemente
printf("\n Maximul este %lf ", max(x,i));
/* functia max e considerata de tip int deoarece nu
exista prototip */
getch();
}
double max(double *tab,int n)
/* functia max (),de tip double; parametrii sunt:
adresa tabloului si numarul de elemente */
{
int i; double vmax=*tab;
for(i=1;i<n;i++)
if(vmax<*(tab+i))
vmax=*(tab+i);
return vmax;
}
```

La compilare, apare mesajul de eroare:

"Type mismatch in redeclaration of max*"*, datorită nepotrivirii tipului din definiție cu cel considerat implicit la apelul funcț iei; soluție:

- definirea funcției înaintea funcției *main* (incomod);
- folosirea prototipului.

Exemplul 1 - varianta 3 - corect----------

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
double max(double tab[],int n);//prototipul
functiei
/* tablou unidimensional pentru care nu trebuie
precizat numarul de elemente;el trebuie cunoscut la
prelucrare si se transfera ca parametru separat */
void main()
{
double x[MAX];
int i;
printf("\n Introd. numerele,terminati cu 0:\n");
for(i=0;i<MAX;i++)
{
scanf("%lf",&x[i]);
if(x[i]==.0) break;
} // la iesire din for, i = nr.elemente
printf("\n Maximul este %lf ", max(x,i));
getch();
}
double max(double tab[],int n)
/* functia max (),de tip double; parametrii sunt:
tabloul tab[] si numarul de elemente */
{
int i; double vmax=tab[0];
for(i=1;i<n;i++)
if(vmax<tab[i])
vmax=tab[i];
return vmax;
}
```

Exemplul 2:

Programul sursă este alcătuit din două fisiere, grupate într-un proiect. Se calculează puterea întreagă a unui număr real.

```
// princ.c
#include <stdio.h>
#include <conio.h>
double power(double, int); // prototip functie
void main()
{
int i; double x;
printf("\n Numarul real: ");
scanf("%lf",&x);
printf(" Puterea intreaga: ");
scanf("%d",&i);
printf("\n Rezultatul este %lf\n", power(x,i));
getch();
}
// func.c
#include <math.h>
double power(double a, int b)
```

```
{
int i, n; double p=1.; n=abs(b);
for(i=1; i<=n; i++)
p*=a;
if(b<0) return (1/p);
else return(p);
}
```

Cele două fisiere se pot compila separat, apoi se grupează într-un proiect, folosindu-se meniul Project al mediului.

Se deschide un fisier proiect (extensia .prj) care va conține numele celor două fisiere; dacă la nume nu se adaugă extensia se consideră că e vorba de fisierele sursa (cu extensia .c); se pot include în proiect direct fisierele *.obj rezultate după compilare.

Pentru a se crea modulul executabil, se selectează "Make EXE file" din meniul Compile, si are loc compilarea (dacă în proiect sunt date fisiere sursă), editarea de legături si crearea fisierului EXE; modulul EXE va avea acelasi nume cu fisierul proiect; se va selecta direct comanda Run pentru crearea fisierului EXE si execuție.

Exemplul 3 - (varianta 1, incorectă)

Programul ilustrează conversia automată de tip de date, la apelul funcției după regulile de conversie cunoscute.

Programul calculează cel mai apropiat număr întreg, pentru un număr real.

```
#include <stdio.h>
#include <conio.h>
void main()
{
float x; printf("\n Introduceti numarul ");
scanf("%f",&x);
printf("\n Cel mai apropiat intreg este %d",
n_int(x));
getch();
}
/* deoarece nu exista prototip, functia n_int este
considerata de tip int iar parametrul i este
convertit automat la double in definitia functiei,
parametrul fiind tip float, apare eroare la
compilare*/
int n_int(float num)
{
if(num>0) {
if(num-(int)num<0.5)
return((int)num); /* se returneaza o valoare
intreaga prin conversie explicita */
else return((int)num+1);
}
else {
if((int)num-num<0.5)
return((int)num);
else
return((int)num-1);
}
}
```

Sunt mai multe puncte de iesire diferite, din funcția n_int dar toate cu acelasi tip de date de iesire.

Exemplul 3 - (varianta 2, corectă)

```
#include <stdio.h>
#include <conio.h>
void main()
{
float x;
printf("\n Introduceti numarul ");
scanf("%f",&x);
printf("\n Cel mai apropiat int este %d",
n_int(x));
// se face conversia la double a lui x
}
int n_int(double num)
{
if(num>0)
{
if(num-(int)num<0.5) return((int)num);
else return((int)num+1);
}
else
{
if((int)num-num<0.5) return((int)num);
else return((int)num-1);
}
}
```

Exemplul 3 - (varianta 3, corectă)

```
#include <stdio.h>
int n_int(float);
// nu e obligatorie prezenta identificatorului de
//parametru, este suficient tipul;
void main()
{
float x;
printf("\n Introduceti numarul ");
scanf("%f",&x);
printf("\n Cel mai apropiat int este %d",
n_int(x));
}
int n_int(float num)
{
if(num>0)
{ if(num-(int)num<0.5) return((int)num);
else return((int)num+1);
}
else{
if((int)num-num<0.5) return((int)num);
else return((int)num-1);
}
```

```
}
```

TEMĂ

1. Să se scrie un program care să citeasca numere reale introduse de la tastatură pâna la citirea numarului 0 si să conțină funcț ii care să calculeze numărul minim, numărul maxim si media numerelor citite.
2. Să se scrie un program care să numere cuvintele dintr-un text de maxim 1000 de caractere de la tastatură. Cuvintele sunt separate prin spațiu, caracterul *Tab* sau *Enter*. Încheierea citirii textului se face prin apasarea tastei *Esc* ('\0x1b').
3. Să se scrie un program care să calculeze si să afiseze inversa unei matrice de 3 linii si 3 coloane cu elemente întregi citite de la tastatură.

2. Transmiterea parametrilor la funcții

Transmiterea parametrilor actuali, către o funcție, se face:
- **prin valoare;**
- **prin referință;**

La transmiterea prin valoare, **se copiază** valoarea fiecărui parametru actual în spațiul rezervat pentru parametrul formal corespunzător. Acest principiu de transfer face ca modificările făcute asupra parametrilor formali, în corpul funcț iei, să nu se reflecte în afara ei, adică **o funcție nu poate modif ica parametrii actuali cu care este apelată**, sau altfel spus, parametrii actuali sunt pentru funcție niste constante predefinite. Acest tip de transfer este predefinit în C si nu poate fi modificat !

Transmiterea parametrilor actuali **prin referință,** se bazează pe tipul de date pointer (adresă). Parametrii actuali ai funcției sunt declaraț i în mod explicit, **pointeri**. În acest tip de transfer, o funcție **primeste indirect valorile actuale** (prin adresele lor) si **le poate modifica** tot în mod indirect, prin intermediul pointerilor de acces, de care dispune.

Exemplu:

Pentru modificarea unor variabile, acestea nu se transferă direct ci prin pointeri, o funcț ie neputând modifica direct valorile variabilelor din funcția care o apelează.

```
#include <stdio.h>
#include <conio.h>
void main()
{
void schimba(int *,int *); /* prototipul folosit
in main, local */
int x=2003, y=0;
printf("\n Valori initiale x=%d y=%d", x, y);
schimba(&x, &y);
printf("\n Valori finale x=%d y=%d", x, y);
getch();
}
void schimba(int *a,int *b)
{
int temp;
temp=*a;
```

```
*a=*b;
*b=temp;
}
```

Programul produce pe ecran:

```
Valori initiale: x=2003 y=0
Valori finale: x=0 y=2003
```

În general, modul în care se transmit parametrii la funcții este dependent de implementare. Unele principii generale, sunt însă valabile la toate implementările. Parametrii sunt transmisi prin stivă. Ordinea de plasare a parametrilor actuali în stivă este, în general, de la dreapta la stânga, pentru lista de parametri din apelul funcției. Standardul nu garantează însă acest lucru, si nici ordinea de evaluare a parametrilor. Un program corect conceput, nu trebuie să depindă de această ordine. Descărcarea stivei se face, de regulă, la majoritatea implementărilor de către funcția apelantă.

Regulile de mai sus sunt determinate de faptul că în C pot exista funcții cu număr variabil de parametri (aceeasi funcț ie poate fi apelată o dată cu *m* parametri si altă dată cu *n* parametri etc.).

Un exemplu clasic este funcția printf(), care are în mod obligatoriu numai primul parametru si anume adresa sirului care descrie formatul, restul parametrior fiind în număr variabil si pot chiar lipsi. Modul de plasare asigură că, totdeauna, funcția printf() va găsi în vârful stivei adresa sirului de caractere care descrie formatul. Analizând specificatorii de format din acest sir, funcția printf() stie câți parametri să-si extragă din stivă si ce dimensiune are fiecare parametru (2 octeți, 4 octeți etc.).

De exemplu, la un apel de forma:

```
printf("%d %s\n", i, s);
```

în vârful stivei se va găsi adresa sirului constant `"%d %s\n"` .

Funcția preia acest parametru din vârful stivei, citeste sirul si extrage din stivă încă doi parametri: un întreg si adresa unui sir de caractere. Evident, dacă parametrii actuali nu concordă (ca număr si tip) cu specificatorii de format, rezultatele vor fi eronate. Este clar acum, de ce este necesar ca stiva să fie descărcată de către programul apelant: acesta stie câți octeți a încărcat în stivă la apelarea unei funcții. De exemplu, un apel de forma:
`printf("%d %d", n);` va produce o a doua tipărire, fără sens (se va tipări, ca întreg, conținutul vârfului stivei înainte de apelul funcției), dar nu va duce la situații de blocare sau pierdere a controlului; revenirea în programul apelant se face corect, iar acesta descarcă stiva de asemenea corect.

Un apel de forma:

```
printf("%d", m, n);
```

va tipări numai variabila *m*, ignorând pe *n* iar restul acțiunilor se execută corect (revenire, descărcare stivă).

3. Variabile globale si locale. Domeniu de valabilitate

La dezvoltarea unui program este util ca problema să fie descompusă în părţi mai simple care să fie tratate separat (pe module). Dezvoltarea modulară a programelor este avantajoasă si conduce la programe structurate. Instrumentul prin care se poate aborda programarea structurată este funcţia. Funcţiile care compun programul se pot afla în unul sau mai multe fisiere sursă, care pot fi compilate separat si încărcate împreună. Este strict necesar săe transmită corect informaţiile între modulele de program.

- Anumite informaţii (valori de variabile) sunt utilizate numai în interiorul funcţiei; ele se numesc **variabile locale** sau **interne** si sunt definite si vizibile în interiorul funcţiei.

- Alte informaţ ii sunt necesare pentru mai multe funcţii si se numesc **globale**; ele se definesc în afara oricărei funcţ ii iar funcţiile au acces la acestea prin intermediul numelui variabilei. O variabilă globală poate fi accesată într-un singur fisier (în care este definită) sau în mai multe fisiere.

- Variabilele globale, fiind accesibile mai multor funcţii, pot transmite informaţ ii între funcţ ii, fiind utilizate ca parametri actuali ai funcţiilor.

3.1. Variabile locale

Sunt definite în interiorul unei funcţii, pot fi utilizate si sunt vizibile numai în interiorul funcţ iei. Nici o altă funcţie nu are acces la variabilele locale, deoarece la iesirea din funcţie, valorile lor sunt distruse (se eliberează memoria ocupată de acestea). Nu există nici o legătură între variabilele cu acelasi nume din funcţii diferite.

Limbajul C nu permite s*ă *se defineas*ă *o funcţie în interiorul altei funcţii .

Cea mai mică unitate de program, în care se poate declara o variabilă locală, este blocul = o instrucţiune compusă, adică o secvenţă de instrucţiuni delimitată de acolade. Limbajul C permite ca într-o instrucţiune compusă să se introducă declaraţii. O funcţie poate conţine mai multe blocuri.

Durata de viaţă a unei variabile este intervalul de timp în care este păstrată în memorie.

- Unele variabile pot rămâne în memorie pe toată durata execuţiei programului, la aceeasi adresă; acestea sunt definite cu cuvântul cheie STATIC - **variabile statice sau permanente** .

- Alte variabile sunt definite si primesc spaţiu de memorie numai când se execută blocul sau modulul în care au fost definite. La încheierea execuţiei blocului sau modulului, se eliberează automat memoria alocată si variabilele dispar. Dacă se revine în blocul respectiv, variabilele primesc din nou spaţ iu de memorie, eventual la alte adrese. Variabilele de acest tip, sunt numite **variabile cu alocare automat**ă a memoriei. Exemplu:

```
float functie()
{
int k ;
static int a[]={1, 2, 3, 4};
. . . . . .
}
```

Variabilele k si a[] sunt locale; domeniul lor de valabilitate este blocul în care au fost definite.

Tabloul a[] păstrează adresa de memorie pe toată durata execuției programului.

Variabila k este cu alocare automată a memoriei.

3.1.1. Inițializarea variabilelor locale

La fiecare apel, variabilele cu alocare automată trebuie inițializate. Dacă nu se face inițializarea, acestea vor avea valori întâmplătoare, care se află în spațiul de memorie alocat. Variabilele statice se inițializează o singură dată. Exemple:

```
void incrementare(void)
{
int i=1;
static int k=1;
i++ ; k++ ;
printf("i=%d \t k=%d \n", i, k);
}
int main(void)
incrementare();
incrementare();
incrementare();
}
```

Rezultatul execuției programului este:

```
i=2 k=2
i=2 k=3
i=2 k=4
```

Variabila i este cu alocare automată; ea este inițializată la fiecare apel al funcției cu valoarea i=1.

Variabila k este de tip **static**, permanentă; se inițializează o singură dată iar valoarea curentă se păstrează la aceeasi adresă.

Variabilele statice dacă nu sunt inițializate, primesc valoarea zero.

Variabilele statice interne (locale) oferă posibilitatea păstrării unor infoemații despre funcție (de exemplu, de câte ori a fost apelată).

3.2. Variabile globale

După domeniul de valabilitate, variabilele pot fi ierarhizate astfel:

- variabile cu acțiune în bloc (locale);
- variabile cu acțiune în funcție (locale);
- variabile cu acțiune în fisierul sursă;
- variabile cu acțiune extinsă la întregul program.

Pentru ca o variabilă să fie valabilă în tot fisierul, trebuie declarată în afara oricărei funcț ii, precedată de cuvântul STATIC. Dacă fisierul conține mai multe funcții, variabila este vizibilă în toate funcțiile care urmează declarației.

Prin convenție, numele unei variabile globale se scrie cu literă mare.

Sfera de acțiune cea mai mare pentru o variabilă este întregul program; o asemenea variabilă este vizibilă în toate funcțiile aflate atât din fisierul în care a fost definită, cât si din alte fisiere.

Pentru a crea o variabilă globală ea se declară în afara oricărei funcții, fără cuvântul cheie STATIC.

```
float x ; //var. globala in tot programul
static float y ; //var. globala in fisierul curent
int main(void)
{
. . . . . .
}
```

Variabilele globale măresc complexitatea unui program deoarece ele pot fi modificate de orice funcție. Se recomandă utilizarea lor numai când este strict necesar.