Considerații teoretice

Cerințe pentru finalizare

Pe măsură ce aplicațiile devin mai complexe, este necesar ca sarcinile de rezolvat să poată fi împărțite în subsarcini pentru ca acestea să poată fi implementate mai ușor. Pentru aceasta, limbajul C oferă posibilitatea de a implementa diverse sarcini în **funcții** separate, fiecare funcție realizând o anumită operație. O funcție are următoarea sintaxă:

```
tip_returnatnume_funcție(param1, param2, ..., paramN)
{
    instrucțiuni
}
```

Cu următoarele componente:

tip_returnat – tipul returnat de funcție (ex: *int*, *float*). Dacă funcția nu returnează nicio valoare, tipul returnat va fi *void* („nimic”).

nume_funcție – numele funcției, analog cu numele unei variabile

param1, …, paramN – parametrii funcției, fiecare dat ca o definire separată de variabilă (ex: *int a*, *float c*). La parametrii unei funcții este nevoie ca fiecare parametru să aibă propriul său tip, chiar dacă e identic cu cel anterior (nu se acceptă „*int a,b*”). În C nu se pot da valori inițiale parametrilor (nu se acceptă „*int a=3*”). Parametrii se pun între paranteze. Chiar dacă funcția nu are niciun parametru, tot trebuie să existe parantezele de parametri, scrise ca „()”.

{…instrucțiuni…} – corpul funcției. Constă dintr-o serie de instrucțiuni puse între acolade „{}”.

O definiție de funcție corespunde notației matematice:

$r=\text{nume\_funcție}(p_1,p_2,\ldots,p_N)$, cu $r \in$ tip_returnat, $p_1 \in T_1$, $p_2 \in T_2$, $\ldots, p_N \in T_N$ unde $T_1,T_2,\ldots,T_n$ sunt tipurile lui $p_1,p_2,\ldots p_N$

Exemplu: Se cere un număr impar n>4. Să se deseneze litera „E” în așa fel încât pe verticală și pe orizontală să fie câte *n* steluțe.

Varianta 1: Rezolvarea acestei probleme **fără funcții** arată ca în programul următor:

```
#include <stdio.h>
int main()
{
    int n, i;
    printf("n=");
    scanf("%d", &n);
     for (i = 0; i < n; i++)                          // @1 -
linia orizontala de sus
        printf("*");
    printf("\n");
     for (i = 0; i < (n - 3) / 2; i++)       // @2 - linia
verticala superioara
        printf("*\n");
     for (i = 0; i < n; i++)                          // @3 -
linia orizontala de la mijloc
        printf("*");
    printf("\n");
     for (i = 0; i < (n - 3) / 2; i++)       // @4 - linia
verticala inferioara
        printf("*\n");
     for (i = 0; i < n; i++)                          // @5 -
linia orizontala de jos
        printf("*");
    printf("\n");
    return 0;
}
```

Se poate constata faptul că există două secvențe de cod care se repetă: trasarea unei linii orizontale se repetă la liniile @1,3,5 și trasarea unei linii verticale se repetă la liniile @2,4. Avem astfel două secvențe de cod duplicate, una de 3 ori, alta de 2 ori.

Duplicarea de cod într-o aplicație este în mod ferm interzisă de normele de practică în programare.

Următoarele două motive sunt evidente:

dacă avem de modificat ceva, atunci acea modificare trebuie realizată în toate secvențele duplicate.

dacă într-o asemenea secvență există o eroare, atunci eroarea va fi prezentă în toate duplicatele. Când se dorește remedierea erorii, remedierea trebuie și ea propagată în toate duplicatele.

Când programele sunt mici, aceste aspecte nu sunt foarte importante. Când dimensiunea programelor crește și codul este scris în mai multe fișiere, este foarte posibil să uităm să modificăm în toate locurile unde apare duplicată acea secvență, sau este posibil ca la unul dintre duplicate să se strecoare o greșeală la modificare. În această situație în aplicație vor apărea erori (bugs) care în general sunt greu de localizat și de reparat.

Duplicarea de cod se rezolvă prin **factorizare**, extrăgând codul comun și implementându-l cu ajutorul unei funcții, funcție care ulterior va putea fi apelată din toate punctele din program de unde este nevoie. Prin această metodă, cele două deficiențe de mai sus dispar:

dacă avem de modificat ceva, vom modifica într-un singur loc (în corpul funcției)

depanarea erorilor se face și ea mai simplu, testând și modificând doar codul funcției

Un alt avantaj major al funcțiilor este faptul că în timp, pe măsură ce implementăm diverși algoritmi, vom avea o bibliotecă de funcții bine puse la punct. Vom putea refolosi aceste funcții și în proiecte viitoare, ceea ce va reduce considerabil timpul de dezvoltare al unei aplicații.

Limbajul C dispune de o colecție bogată de funcții standard (C standard library). Acestea sunt grupate pe categorii și sunt accesibile prin intermediul includerii unor fișiere antet (header, .h). Printre funcțiile standard se găsesc funcții pentru fișiere, pentru caractere sau șiruri de caractere, matematice, formatare, gestiunea memoriei, etc. Este bine să se parcurgă aceste funcții pentru a ne

forma o idee despre ce ni se pune la dispoziție și deci ce putem folosi fără a mai fi nevoie ca noi să implementăm de la zero.

Pe lângă funcțiile C standard, mai putem găsi biblioteci de funcții pentru cele mai variate aplicații: grafică, rețele de calculatoare, simulare, baze de date, etc. Multe dintre aceste biblioteci au fost dezvoltate ani de zile de către echipe de programatori și implementează funcționalități complexe, astfel încât folosirea lor ne poate scuti de foarte mult efort la scrierea aplicației noastre.

Varianta 2: rezolvarea problemei anterioare **folosind funcții**:

```
#include <stdio.h>
void linieOrizontala(int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("*");
    printf("\n");
}

void linieVerticala(int n)
{
    int i;
    for (i = 0; i < (n - 3) / 2; i++)
        printf("*\n");
}

int main()
{
    int n;
    printf("n=");
    scanf("%d", &n);
    linieOrizontala(n);              // linia orizontala
de sus
      linieVerticala(n);                        // linia
verticala superioara
    linieOrizontala(n);              // linia orizontala
de la mijloc
      linieVerticala(n);                        // linia
verticala inferioara
    linieOrizontala(n);              // linia orizontala
de jos
```

```
    return 0;
}
```

În această variantă s-au folosit două funcții:

linieOrizontala - pentru trasarea unei linii orizontale

linieVerticala - pentru trasarea unei linii verticale

Fiecare funcție primește un singur parametru „*int n*” și nu returnează nicio valoare (sunt de tip „**void**”). Se poate constata că fiecare funcție este destul de simplă și ușor de citit, deoarece conține doar o secvență cu funcționalitate specifică de cod. În același timp și programul principal, *main*, este mult mai simplu și mai lizibil, deoarece dacă funcțiile au nume bine alese, programatorul are o imagine intuitivă asupra rolului lor.

Și în această a doua variantă apare o ușoară duplicare de cod: secvența *linieOrizontală/linieVerticală* se repetă de două ori. Putem elimina și această duplicare, implementând o funcție ”*linieOV(int lungime, int inaltime)*” ceea ce ne-ar ajuta și la scrierea altor litere, gen L, F, C.

În C, fiecare funcție trebuie scrisă în afara altei funcții, deci nu am putut să scriem cele două funcții auxiliare în interiorul lui *main*. La fel ca și în cazul variabilelor, **o funcție trebuie mai întâi definită și abia apoi folosită.** Din acest motiv, dacă am fi implementat cele două funcții după *main*, am fi avut eroare de compilare în *main*, fiindcă nu s-ar fi știut ce înseamnă *linieOrizontala* și *linieVerticala*. De aceea, funcțiile au trebuit implementate **înainte** de a fi folosite.

Funcția **main** este o funcție obișnuită ca sintaxă, dar compilatorul o interpretează ca fiind **începutul** programului, chiar dacă înaintea ei mai sunt și alte funcții. Din acest motiv, întotdeauna un program C începe cu funcția *main*, oriunde ar fi aceasta. Valoarea returnată de funcția *main* este considerată de către sistemul de operare (Windows, Linux, etc) un indicator al felului în care s-a executat aplicația: cu succes (valoarea returnată 0) sau cu eroare (altă valoare).

Dacă o variabilă se declară într-o funcție (ex: *i* din funcția *linieOrizontala*), atunci acea variabilă este **locală** funcției respective și ea nu există decât în interiorul acelei funcții (este vizibilă doar în interiorul funcției). La fel și parametrii unei funcții sunt locali funcției în care au fost declarati. Mai multe funcții pot să aibă aceleași nume pentru parametri și pentru variabile locale (ex: *linieOrizontala* și *linieVerticala* au fiecare definită câte o variabilă *i*), fără ca cele două variabile să se suprapună sau să existe vreo problemă în selectarea lor de către compilator.

Dacă o variabilă este definită în afara unei funcții, ea este **globală**. Iată un tabel comparativ între variabilele locale și cele globale:

Variabile local	Variabile globale
Nu sunt inițializate de compila au valori nedeterminate.	Sunt inițializate chiar de la începutul p valoarea 0 (depinde de compilator)
Există (ocupă memorie) doar apelului funcției.	Există de la începutul până la sfârșitul
Sunt vizibile (accesibile) doar care sunt declarate.	Sunt vizibile din tot programul.

O funcție poate returna o valoare sau poate fi întreruptă din execuție folosind instrucțiunea **return**:

```
return expresie;//pentru funcții care returnează o valoare
return;//pentru funcții void
```

return poate să apară oriunde în corpul unei funcții, nu doar la sfârșit. El are ca efect terminarea imediată a funcției respective și revenirea în programul apelant.

Transmiterea argumentelor

Când se apelează o funcție, valorile expresiilor date ca argumente se vor copia în parametrii funcției și apoi funcția va folosi aceste copii ale valorilor originare. Acest mod de apel se numește **transmitere prin valoare**. Din acest motiv, chiar dacă modificăm valoarea unui parametru în interiorul funcției, această modificare se

efectuează doar pe copia valorii cu care a fost inițializat acel parametru, fără ca valoarea originară să fie afectată.

Exemplu: Să se scrie o funcție *swap(x,y)*, care să interschimbe valorile variabilelor date ca parametri.

Varianta 1: *swap1* **nu** implementează corect cerința din exemplu!!!

```
void swap1(int x,int y)
{
    int tmp=x;
    x=y;
    y=tmp;
}

int main()
{
    int a=5,b=7;
    swap1(a,b);
    printf("%d %d\n",a,b);// 5 7
    return 0;
}
```

Într-o primă variantă încercăm să scriem o funcție *swap1*, care are 2 parametri pe care-i interschimbă. Apelăm această funcție cu două variabile *a* și *b*. Ne-am aștepta ca după apelul funcției, valorile lui *a* și *b* să fie interschimbate, iar pe ecran să se afișeze ”7 5”. Cu toate astea, se vor afișa tot valorile inițiale, deci funcția *swap1* nu le-a interschimbat.

De ce *swap1* nu a interschimbat valorile lui *a* și *b*? Răspunsul constă chiar în transmiterea parametrilor prin valoare. În acest caz, la apelul lui *swap1(a,b)*, valorile lui *a* și *b* au fost copiate în *x* și *y*. Ulterior, în timpul execuției lui *swap1*, când *x* și *y* au fost interschimbate, de fapt s-au interschimbat valorile din aceste două variabile, fără ca originalele *a* și *b* să fie afectate în vreun fel.