Programarea calculatoarelor
Limbajul C

CURS 13

Programe complexe
Directive preprocesare
Convenții programare

Programe complexe. Compilări separate. Fişiere proiect.

- Funcțiile unei aplicații complexe se pot afla în mai multe fişiere sursă, compilate separat şi legate împreună într-un singur program executabil.

- Modificări ale programului se vor face prin editarea şi recompilarea unui singur fişier sursă (sau a câtorva fişiere) şi nu a întregului program (se evită recompilarea unor funcții care nu au suferit modificări).

Programe complexe. Compilări separate. Fişiere proiect.

- Este posibilă dezvoltarea şi testarea în paralel a unor funcții din aplicație de către persoane diferite.
- Specificarea fişierelor obiect (OBJ) care vor fi legate într-un singur executabil se face printr-un fişier proiect atunci când se lucrează cu un mediu integrat (IDE) cum este Dev-C++.
- In forma sa cea mai simplă un fişier proiect este o listă de nume de fişiere sursă (C sau CPP) şi/sau fişiere obiect (OBJ) şi/sau biblioteci (LIB) care contribuie la producerea unui fişier executabil.

Etape elaborare

- înțelegerea specificațiilor problemei de rezolvat şi analiza unor produse software asemănătoare
- stabilirea funcțiilor de bibliotecă care pot fi folosite şi verificarea modului de utilizare a lor (pe exemple simple)
- determinarea structurii mari a programului: care sunt principalele funcții din componența programului şi care sunt eventualele variabile externe

Exemplu

- program care să realizeze efectul comenzii DIR din MS-DOS (“dir” şi “ls” din Linux): să afişeze numele şi atributele fişierelor dintr-un director dat explicit sau implicit din directorul curent.
- va conține cel puțin trei module principale :
  - preluare date inițiale (“input”),
  - obținere informații despre fişierele cerute (“getfiles”)
  - prezentarea acestor informații (“output”), plus un program principal.
- module realizate ca fişiere sursă separate

Dezvoltarea în etape

- Definirea progresivă a funcțiilor din componența aplicației, fie de sus în jos (“top-down”), fie de jos în sus (“bottom-up”), fie combinat.

- Abordarea “bottom-up “
  - definirea unor funcții mici, care vor fi apoi apelate în alte funcții, s.a.m.d. până se ajunge la programul principal.

Dezvoltarea în etape

- Abordarea “top-down”
  - stabileşte funcțiile importante şi programul principal care apelează aceste funcții.
  - se defineşte fiecare funcție, folosind eventual alte funcții încă nedefinite, dar care vor fi scrise ulterior.

Exemplu:

```
void  main(int argc, char * argv[]) {
   char  *files[MAXF];     // vector cu nume de fişiere
   int nf;                      // numar de fişiere
   getargs (argc,argv);   // preluare date
   nf=listFiles(files);       // creare vector de fişiere
   printFiles(files,nf);     // afisare cu atribute
}
```

Compilări separate

- Compilarea separată a unor părți din programele mari:
  - Enumerarea modulelor obiect şi bibliotecilor statice componente.
  - Descrierea dependențelor dîntre diverse fişiere (surse, obiect, executabile) astfel ca la modificarea unui fişier să se realizeze automat comenzile necesare pentru actualizarea tuturor fişierelor dependente de cel modificat.

Dezvoltare programe complexe. Compilări separate

- Dezvoltarea de programe C în mod linie de comandă:
  - Enumerarea fişierelor obiect şi bibliotecilor în comanda de linkeditare.

    Exemplu:

    BCC M.CPP F1.CPP F2.CPP
  - Utilizarea unui program de tip “make” şi a unor fişiere ce descriu dependențe între fişiere şi comenzi asociate (“makefile”).
- Când se foloseşte un mediu integrat pentru dezvoltare (IDE) soluția o constituie fişierele proiect.

Fişiere antet (header)

- Funcțiile unei aplicații pot folosi în comun:
  - tipuri de date definite de utilizatori
  - constante simbolice
  - variabile externe
- Elementele comune se definesc de obicei în fişiere antet (de tip H), care se includ în compilarea fişierelor sursă cu funcții (de tip C sau CPP).
- Tot în aceste fişiere se declară funcțiile folosite în mai multe fişiere din componența aplicației.

Fişiere antet (header) – conținuare

- Exemplu:

```
#define MAXC 256    // dimensiunea unor siruri
#define MAXF 1000  // numar de fişiere estimat
struct file {
 char fname[13];      // nume fişier (8+3+'.'+0)
 long fsize;          // dimensiune fişier
 char ftime[26] ;     // data ultimei modificari
 short isdir;         // daca fişier director
};
int f(int, float);
```

- Fişierul antet "dirlist.h" poate include şi fişiere antet standard comune ("stdio.h", stdlib.h"); este însă posibil ca includerile de fişiere antet standard să facă parte din fiecare fişier sursă al aplicației.

Directive de preprocesare

- Suprimarea unei definiții
  #undef identificator
- Compilări condiționate de expresii
  #if expresie_constanta
  else
  #endif
- Compilări condiționate de definire constante simbolice
  #ifdef NUME
  #endif

  #ifndef NUME
  #endif
- Directivele din grupul *if* au mai multe forme, iar un bloc *if ... endif* poate conține şi o directivă *elseif*

Programe complexe. Compilări separate. Fişiere proiect.

- Pentru a evita includerea multiplă a unui fişier antet (standard sau nestandard) se recomandă ca fiecare fişier antet să înceapă cu o secvență de felul următor:

```
#ifndef  HDR
    #define HDR
      // conținut fişier HDR.H ...
#endif
```

- O soluție alternativă este ca în fişierul ce face includerea să avem o secvență de forma următoare:

```
#ifndef STDIO_H
    #include <stdio.h>
    #define STDIO_H
#endif
```

Clasa Extern **de alocare a memoriei**

- o variabila sau o funcție declarata extern este vizibilă şi în alt fişier decat cel în care este definită
- permite referirea unei variabile *globale* definită în afara unității de program.

```
/* fişierul main.c */
#include"header.h"

int a = 1, b = 2, c = 3; /* variabile globale */
int f(void); /* prototip */

int main(void){
    printf("a = %d, b = %d, c = %d, f = %d\n", a, b, c, f());
    system("pause");
    return 0;
}
```

Clasa Extern **de alocare a memoriei**

```
/* fişierul funcția.c */
int f(void){
     extern int a; /* cauta a în afara fişierului */
     int b, c; /* b, c locale */
     b = c = 22;
     return (a + b + c);
}

/* fişierul antet.h */
#ifndef  HDR
   #define HDR
   #include<stdio.h>
   #include<stdlib.h>
#endif
```

Variabile externe

- Se definesc într-unul din fişierele sursă ale aplicației

```
char path[MAXC], mask[MAXC], opt[MAXC];  // var comune
```

- Domeniul implicit al unei variabile globale este fişierul în care variabila este definită (mai precis, funcțiile care urmează definiției).
- Declarare variabilă cu atributul extern, în toate fişierele în care se fac referiri la ea.
- Exemplu :

```
extern char path[MAXC], mask[MAXC], opt[MAXC];
```

Variabilele externe – continuare

- Pentru a evita declarațiile extern în toate fişierele sursă, mai putin fişierul ce conține definițiile variabilelor externe:
  - Se defineşte în fişierul sursă cu definițiile variabilelor externe un nume simbolic oarecare:

    #define MAIN
  - In fişierul “dirlist.h” se plasează toate declarațiile de variabile externe, încadrate de directivele if şi endif:

    #ifndef MAIN

    extern char path[MAXC], mask[MAXC], opt[MAXC];

    #endif

Best practices: Foarte important!

- Dezvoltarea progresivă a programelor, cu teste cât mai complete în fiecare etapă!
- Păstrarea versiunile corecte anterioare, chiar incomplete, pentru a putea reveni la ele dacă prin extindere se introduc erori sau se dovedeste că soluția de extindere nu a fost cea mai bună!
- Comentarea rolului unor variabile sau instrucțiuni se va face chiar la scrierea lor în program şi nu ulterior!

Exemplu

1. O mulțime de numere poate fi reprezentată printr-o structură care grupează un vector de întregi şi dimensiunea sa. Să se scrie un proiect care conține următoarele:

- Definiția tipului mulțime şi antetele funcțiilor definite - într-un fişier **“set.h”:**

  typedef struct { ... } set;

  etc.

- Funcții pentru operații cu mulțimi (cu argument de tip "set" sau "set*") într-un fişier **“set.c”:**

  - căutare întreg într-o mulțime (cu rezultat 0 sau 1)

  - creare mulțime vidă

Exemplu

- adăugare întreg la o mulțime, dacă nu exista deja
- eliminare întreg dat dintr-o mulțime
- intersecția a două mulțimi
- citirea unei mulțimi
- afişarea unei mulțimi
- test mulțime vidă

- Un program principal care citeşte două mulțimi şi afişează mulțimea intersecție a celor două într-un fişier "test.c".
- Se vor utiliza directive de compilare condiționată pentru evitarea includerii multiple a fişierului header

Rezolvare

```
//Set.h
#ifndef SET_H
	#define SET_H
	#include<stdio.h>
	#include<stdlib.h>

	typedef struct {
		int v[100];
		int n;
	  }Set;

	void init(Set *);
	void add(Set *, int);
	void read(Set *);
	void print(Set);
	int empty (Set);
	int search(Set , int);
	void del(Set *, int);
	Set intersect(Set , Set);
#endif
```

Rezolvare

```
//Set.c
#include "set.h"
int search (Set a, int x) {
     int i;
     for (i=0; i<a.n; i++)
            if ( a.v[i] == x ) return i;
     return -1;
}
void init (Set *a) {
     a->n = 0;
}
void add (Set *a, int x) {
     a->v[a->n++] = x;
}
int empty (Set a){
    return a.n==0;
}
```

Rezolvare

```
void read (Set *a){
      int i;
      printf ("Numar elemente mulțime:\n");
      scanf ("%d", &a->n);
      printf("Elemente mulțime:\n");
      for (i=0; i<a->n; i++)
              scanf ("%d", &a->v[i]);
}
void print (Set a){
      int i;
      printf ("Numar elemente mulțime: %d\n", a.n);
      if(!empty(a)){
        printf ("Elemente mulțime:\n");
        for (i=0; i<a.n; i++)
             printf ("%5d", a.v[i]);
        printf ("\n");
     }
}
```

Rezolvare

```
void del (Set *a, int x){
    int i, j;
    if ( (i=search(*a,x)) != -1){
        for (j=i; j<a->n-1; j++) a->v[j] = a->v[j+1];
        a->n--;
    }
}
Set intersect(Set a, Set b){
    int i;
    Set c;
    init(&c);
    for (i=0; i<a.n; i++)
        if ( search(b,a.v[i])!= -1) add(&c, a.v[i]);
    print(c);
    return c;
}
```

Rezolvare

```
//Test.c
#include"set.h"
int main(){
    Set a, b;
    read(&a);
    read(&b);
    print(intersect(a,b));
    system("pause");
    return 0;
}
```

Convenții de programare

Beneficii:

- Uniformizează modul de scriere a codului
- Facilitează citirea şi înțelegerea unui program
- Facilitează întreținerea aplicațiilor
- Facilitează comunicarea între membrii unei echipe ceea ce duce la un randament sporit al lucrului în echipă

Obs:

- Permite manifestarea creativității programatorului
- Poate fi imbunătățit şi adoptat ca standard

Convenții de programare: Variabile

- Toate variabilele se definesc înainte de partea de cod propriu-zis (instrucțiuni)
- Variabilele locale se definesc câte una pe linie; excepție fac indecşii, variabilele temporare şi variabilele inițializate cu aceeaşi valoare

```
int  Zone;
int  i, j, counter;
int  Mode = k = 0;
```

- Variabilele locale ar trebui inițializate înainte de utilizare
- Se va evita utilizarea variabileleor globale pe cât posibil

Convenții de programare: Dimensiune vectori

- Nu se transmite dimensiunea maximă a unui vector când acesta este parametru al unei funcții; aceasta se transmite separat într-o variabilă

Nerecomandat

```
char *substring(char string[80], int start_pos, int length)
{…
}
```

Recomandat

```
char *substring(char string[], int start_pos, int length)
{…
}
```

Convenții de programare: Identificatori

- Identificatorii trebuie să îndeplinească standardul ANSI C (lungimea<31, caractere permise: litere, cifre, _)
- Simbolul '_' nu va fi folosit ca prim caracter al unui identificator
- Nu se vor folosi nume utilizate de sistem decât dacă se doreşte înlocuirea acestora (constante, fişiere, funcții)
- Toate fişierele header vor avea extensia ".h"
- Toate constantele simbolice definite cu #define vor fi scrise cu litere mari
- Numele unui fişier nu trebuie să depşească 14 caractere în lungime (cu extensie)

Convenții de programare

- Variabilele locale, definițiile de tip (typedef), numele de fişiere cat şi membrii unei structuri vor fi scrişi cu litere mici
- Numele variabilelor şi funcțiilor trebuie să fie semnificative şi în concordanță cu ceea ce reprezintă
  - Contor, index: g, h, i, j, k, l
  - Pointeri: ptr_var, var_ptr, var_p, p_var, p
  - Variabile temporare: tmp_var, var_tmp, t_va
  - Valori returnate: status, return_value
  - Funcții: readValues, reset_status, print_array
  - Fişiere: initialData.txt, entry.txt, setFuncțions.c, set.h
- Variabilele globale trebuie să se diferențieze de cele locale (fie primul caracter literă mare, fie un sufix de genul “global”): Vec_initial, vec_global, set_glb

Convenții de programare: Funcții

- Se vor folosi pe cât posibil funcții standard în loc de funcții specifice sistemului de operare
  - System Dependent
    open(), close(), read(), write(), lseek(), etc...
  - ANSI Funcțions
    fopen(), fclose(), fread(), fwrite(), fseek(), etc...
- Toate funcțiile vor avea tip definit explicit
- Funcțiile care întorc pointeri vor returna NULL în cazul neîndeplinirii unei condiții
- Numărul parametrilor unei funcții ar trebui limitat la cinci sau mai puțin
- Toate funcțiile definite trebuie însoțite de antete ce vor conține lista tipurilor parametrilor

Convenții de programare: Constante

- Toate constantele folosite într-un fişier vor fi definite înainte de prima funcție din fişier
- Dacă se definesc constantele TRUE şi FALSE, acestea trebuie să aibă valoare 1, respectiv 0:

```
#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif
```

- Definirea de constante simbolice (#define) va fi preferată utilizării directe a valorilor:

```
#define NMAX 100
int a[NMAX];
```

Convenții de programare: Coding practices

- Se recomandă utilizarea standardului ANSI C pentru portabilitate
- Programele nu trebuie să depindă de caracteristicile compilatorului (ordinea de evaluare a expresiilor
  - k = ++i + i; /*gresit */
  - y = f(x) + z_glb; /*gresit daca f() schimba valoarea lui z_glb*/
  - k = ++i + j++; /* OK */
  - a[i++] = j++; /* OK */
- Orice definire (structura, enumerare, tip, etc) utilizată în mai multe fişiere va fi inclusă într-un fişier antet (.h)
- Toate conversiile de tip vor fi făcute explicit
- Variabilele structura se vor transmite prin adresa

Convenții de programare

- Pentru constantele utilizate pentru activarea / dezactivarea unor instrucțiuni se va verifica definirea acestora utilizând compilările condiționate
  - **Nerecomandat**

```
#define DEBUG 4 /* used for level of debugging desired */
for (i = 0; i < 5 && DEBUG; i++)
{
   printf("i = %d\n", i);
}
```

  - **Recomandat**

```
#define DEBUG
#ifdef DEBUG
for (i = 0; i < 5; i++)
{
   printf("i = %d\n", i);
}
#endif
```

Convenții de programare

- Pentru un simbol testat cu #ifdef, #ifndef sau #if defined nu se va defini o valoare

Nerecomandat

```
#define DEBUG 0
#ifdef DEBUG
for (i = 0; i < 5; i++)
{
   printf("i = %d\n", i);
}
#endif
```

Recomandat

```
#define DEBUG
#ifdef DEBUG
for (i = 0; i < 5; i++)
{
   printf("i = %d\n", i);
}
#endif
```

Convenții de programare

- Elementele unui vector vor fi accesate utilizând [] şi nu operatorul de dereferențiere *.

  Nerecomandat

  int array[11];

  *(array + 10) = 0;

  Recomandat

  int array[11];

  array[10] = 0;

- Transmiterea parametrilor prin pointeri va fi evitată ori de cate ori este posibil

  x = f(a, b, c) şi x = f(a, b, c, x)

  sunt mai uşor de înțeles decât

  f(a, b, c, &x)

Convenții de programare

- Toate instrucțiunile *switch* vor avea clauza “default” care întotdeauna va fi ultima

```
switch (int_variable)
{
    case:..
            break;
    case:..
            break;
    default:..
            break;
}
```

- Operatorul “,” va fi utilizat doar în instrucțiunea *for* şi la declararea variabilelor
- Modificarea unui cod existent se va face conform standardului existent deja în acel cod

Convenții de programare

- Modulele unui program nu trebuie să depăşească un anumit grad de complexitate şi un anumit număr de linii (maxim o jumătate de pagina)
- Se va utiliza evaluarea condiției afirmative mai degrabă decât a celei negative (!)
- Condițiile logice vor fi scrise explicit

Nerecomandat

```
if (is_available)
if (sys_cfg__is_radio_retry_allowed())
if (intermediate_result)
```

Recomandat

```
if (is_available == FALSE)
if (sys_cfg__is_radio_retry_allowed() == TRUE)
if (intermediate_result != 0)
```

Convenții de programare

- Nu se recomandă utilizarea variabilelor globale; daca vor fi utilizate trebuie indeplinite următoarele cerințe:
  - Toate variabilele globale pentru un proiect vor fi definite într-un singur fişier
  - Variabilele globale vor fi inițializate înainte de utilizare
  - O variabilă globală va fi definită o singură dată pentru un program executabil

Convenții de programare: Includerea fișierelor

- Fişierele antet vor defini o constantă simbolică pentru a permite includerea multiplă. Dacă fişierul se numeşte file.h constanta se poate numi FILE_H
  - "example.h"

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
…
#endif
```

- Se recomandă ca fişierele antet să nu includă alte fişiere antet
- Pentru includerea unui fişier antet definit de utilizator se vor utiliza “ ”, iar pentru o bibiloteca standard < >

Convenții de programare: Macrouri

- In macrourile tip funcție parametrii vor fi scrişi între paranteze
  - **Nerecomandat**

```
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", a,
                                        __LINE__, b);
```

  - **Recomandat**

```
#define prt_debug(a, b) printf("ERROR: %s:%d, %s\n", (a),
                                        __LINE__, (b));
```

- Macrourile complexe vor fi comentate
- Un macrou nu va depăşi 10 linii

Convenții de programare: Stil de programare

- Se va scrie o singură instrucțiune pe linie
- Tipul unei funcții şi numele acesteia vor fi pe aceeaşi linie

Nerecomandat

```
int
err_msg(int error_code)
{
    …
}
```

Recomandat

```
int err_msg(int error_code)
{
    …
}
```

Stil de programare - Spațierea

1. Nu vor fi spații albe:

- dupa un cast explicit de tip
  - **Nerecomandat**

    int x = 1;

    double y = 3.0;

    y = (double) x + 16.7;
  - **Recomandat**

    int x = 1;

    double y = 3.0;

    y = (double)x + 16.7;
- Între operatorii unari (&, *, -, ~, ++, --, !, cast, sizeof) şi operanzii lor
- Înainte sau după operatorii primari (“()”,”[]”,”.”,”->”)

Stil de programare - Spațierea

2. Nu vor fi spații albe:

- Între caracterul # şi directiva de preprocesare
  - **Nerecomandat**
    - #
    - define TEST 0

    sau
    - # define TEST 0
  - **Recomandat**
    - #define TEST 0
- Între numele unei funcții şi paranteza care îi urmează
- între primul argument al funcției şi paranteza deschisă
- între ultimul argument al funcției şi paranteza închisă
  - **Nerecomandat**
    - just_return (arg1, arg2);
  - **Recomandat**
    - just_return(arg1, arg2);
- Între parantezele deschisă, respectiv închisă şi expresia unei instrucțiuni condiționale

  if (x == y)…

Stil de programare - Spațierea

3. Un singur spațiu

- va exista între expresia condițională a unei instrucțiuni şi instrucțiune
  - **Nerecomandat**

    if(x == y)
  - **Recomandat**

    if (x == y)
- precede şi urmează operatorii de atribuire, operatorii relaționali,operatorii logici, operatorii aritmetici (excepție cei unari) operatorii pe biți şi operatorul condițional

  a = 3;
- va urma unei virgule

  int a, b, c;

Stil de programare - Spațierea

- Va exista cel puțin o linie goală care să separe definirea variabilelor locale de instrucțiuni

```
int just_return(int first_arg, int second_arg)
{
    /*-------------- LOCAL VARIABLES -------------*/
    int i = 0;              /* Loop counter. */
    int j = 0;              /* Loop counter. */
    /*-------------------- CODE -----------------*/
    /*
    * Body of funcțion just_return.
    */
    return(0);
}
```

- Membrii unei structuri, uniuni, enumerări vor fi plasați pe linii distincte la declararea lor

Utilizarea acoladelor şi parantezelor

- Componentele logice ale unei expresii condiționale vor fi grupate cu paranteze chiar dacă acestea nu sunt necesare

```
if ((x == y) && (a == b))
```

- Toate acoladele vor fi câte una pe linie

```
for (i = 0; i < loop_cntrl; i++)
{
    /* Body of ‘for’. */
}
```

- Toate instrucțiunile condiționale vor folosi acolade pentru identificarea corpului acestora chiar dacă acesta este format dintr-o singură instrucțiune

```
if (printf_debug_info == TRUE)
{
    printf(“%s”, debug_info);
}
```

Indentarea

- Toate definițiile de funcții încep în coloana 1
- Acoladele ce definesc corpul unei funcții vor fi în coloana 1
- Acoladele corespunzătoare unei instrucțiuni, unei inițializări de structură, vector, etc vor fi în aceeaşi coloană cu instrucțiunea sau inițializarea respectivă

```
for (i = 0; i < loop_cntrl; i++)
{
   /* Body of ‘for’. */
}
```

- Toate blocurile incluse în alt bloc vor fi indentate cu 2 până la 4 spații albe

Indentarea

Vor fi indentate cu 2 - 4 spații:

- câmpurile unui tip de date

```
struct example_type
{
  int x;
  double y;
};
```

- ramurile *case* ale unei instrucțiuni *switch*
- continuarea unei linii, față de operatorul de atribuire sau față de paranteza deschisă în cazul unei instrucțiuni sau a unui apel de funcție
  - num = this_example_test_structure.example_struct_field1 *
    this_example_test_structure.example_struct_field2;
  - if ((very_long_result_variable_name >=
    lower_specification_value))

Indentarea

- Instrucțiunile aflate la acelaşi nivel de includere vor fi indentate la aceeaşi coloană

```
if (condiție1 == TRUE)
{
        /* Body of if. */
}
else
        if
        {
                /* Body of second if. */
        }
        else
        {
                /* Body of else. */
        }
```

Comentarii

- Vor completa codul, nu îl vor dubla!
- Explică mai mult decât este subînțeles din cod
- Nu trebuie să fie foarte multe comentarii (îngreunează codul) dar nici foarte puține (nu este explicat codul)
- Pot fi comentarii bloc, pe o linie, sau in-line

Comentarii

- Comentarii bloc: descriu secțiunile principale ale programului şi vor fi indentate la acelaşi nivel cu codul pe care il comentează

```
/*
*   This is one format which can be
*   used for block comments.
*/
/*********************************************************
*
*   This is also an acceptable format for block comments.
*
*********************************************************/
/*
*********************************************************
*
*   This is also an acceptable format for block comments.
*
*********************************************************
*/
```

Comentarii

- Comentarii pe o linie
  - Se indentează la acelaşi nivel cu codul pe care îl comentează

```
if (argc > 1)
{
    /* Get input file from command line. */
    if ((freopen(argv[1], 'r', stdin) == NULL)
    {
        /* Body of if. */
    }
}
```

- Comentarii in-line (pentru descrierea declarațiilor)
  - Trebuie să fie indeajuns de scurte încât să intre pe aceeaşi linie cu codul comentat

```
int i = 0;              /* Loop counter. */
int status = TRUE;      /* Return value of this funcțion.*/
```

Reguli generale privind comentariile

- Comentariile nu vor fi incluse unele în altele
- Fiecare variabilă locală va avea un comenatriu ce va descrie utilizarea ei dacă aceasta nu reiese din nume
- Comentariile in-line trebuie să fie aliniate pe cât posibil la stânga în cadrul unei funcții

```
j = 5;        /* Assign j to the starting string position */
k = j + 9;    /* Assign k to the ending string position    */
```

- Instrucțiunile condiționale sau buclele complexe (mai mult de 10 linii necomentate) vor avea ataşat un comentariu la acolada de închidere ce va indica închiderea instrucțiunii şi unul la începutul sau în interiorul blocului ce va indica scopul acestuia

```
if (a>b){
   /* purpose
   ....
   ....*/
...
} // end of if(a>b)
```

Exercițiu:

1. Să se rescrie exemplul anterior utilizând convențiile de implementare descrise (identificatori, convenții de stil, etc.)