

# Programarea calculatoarelor

## Limbajul C



### **CURS 12**



Programe complexe  
Directive preprocesare  
Convenții programare



# Programe complexe. Compilări separate. Fișiere proiect.

---

- Funcțiile unei aplicații complexe se pot afla în mai multe fișiere sursă, compilate separat și legate împreună într-un singur program executabil.
- Modificări ale programului se vor face prin editarea și recompilarea unui singur fișier sursă (sau a câtorva fișiere) și nu a întregului program (se evită recompilarea unor funcții care nu au suferit modificări).

# Programe complexe. Compilări separate. Fișiere proiect.

---

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

# Etape elaborare

---

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

# Exemplu

---

- program care să realizeze efectul comenzii DIR din MS-DOS (“dir” și “ls” din Linux): să afișeze numele și attributele fișierelor dintr-un director dat explicit sau implicit din directorul curent.
- va conține cel puțin trei module principale :
  - preluare date inițiale (“input”),
  - obținere informații despre fișierele cerute (“getfiles”)
  - prezentarea acestor informații (“output”), plus un program principal.
- module realizate ca fișiere 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”
  - stabilește funcțiile importante și programul principal care apelează aceste funcții.
  - se definește 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 fișiere  
    int nf; // numar de fișiere  
    getargs (argc,argv); // preluare date  
    nf=listFiles(files); // creare vector de fișiere  
    printFiles(files,nf); // afisare cu attribute  
}
```

# Compilări separate

---

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



# Dezvoltare programe complexe. Compilări separate

---

- Dezvoltarea de programe C în mod linie de comandă:
  - Enumerarea fișierelor obiect și bibliotecilor în comanda de linkeditare.  
Exemplu:  
BCC M.CPP F1.CPP F2.CPP
  - Utilizarea unui program de tip “make” și a unor fișiere ce descriu dependențe între fișiere și comenzi asociate (“makefile”).
- Când se folosește un mediu integrat pentru dezvoltare (IDE) soluția o constituie fișierele proiect.

# Fișiere 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 fișiere antet (de tip H), care se includ în compilarea fișierelor sursă cu funcții (de tip C sau CPP).
- Tot în aceste fișiere se declară funcțiile folosite în mai multe fișiere din componența aplicației.

# Fișiere antet (header) – continuare

---

- Exemplu:

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

- Fișierul antet “dirlist.h” poate include și fișiere antet standard comune (“stdio.h”, “stdlib.h”); este însă posibil ca includerile de fișiere antet standard să facă parte din fiecare fișier sursă al aplicației.

# Directive de preprocesare

---

- Suprimarea unei definiții  
#undef identificador
- 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 și o directivă *elseif*

# Programe complexe. Compilări separate.

## Fișiere proiect.

---

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

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

- O soluție alternativă este ca în fișierul 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 și în alt fișier decat cel în care este definită
- permite referirea unei variabile *globale* definită în afara unității de program.

```
/* fișierul 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

---

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

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

# Variabilele externe – continuare

---

- Pentru a evita declarațiile extern în toate fișierele sursă, mai puțin fișierul ce conține definițiile variabilelor externe:
  - Se definește în fișierul sursă cu definițiile variabilelor externe un nume simbolic oarecare:  
`#define MAIN`
  - În fișierul “dirlist.h” se plasează toate declarațiile de variabile externe, încadrate de directivele if și 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 versiunilor 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 și nu ulterior!

# Exemplu

---

1. O mulțime de numere poate fi reprezentată printr-o structură care grupează un vector de întregi și dimensiunea sa. Să se scrie un proiect care conține următoarele:
  - Definiția tipului mulțime și antetele funcțiilor definite - într-un fișier “**set.h**”:  
typedef struct { ... } set;  
etc.
  - Funcții pentru operații cu mulțimi (cu argument de tip "set" sau "set\*") într-un fișier “**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
- afișarea unei mulțimi
- test mulțime vidă
- Un program principal care citește două mulțimi și afișează mulțimea intersecție a celor două într-un fișier "test.c".
- Se vor utiliza directive de compilare condiționată pentru evitarea includerii multiple a fișierului 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 multime:\n");
    scanf ("%d", &a->n);
    printf("Elemente multime:\n");
    for (i=0; i<a->n; i++)
        scanf ("%d", &a->v[i]);
}

void print (Set a){
    int i;
    printf ("Numar elemente multime: %d\n", a.n);
    if(!empty(a)){
        printf ("Elemente multime:\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));
    return 0;
}
```



# Convenții de programare

---

## *Beneficii:*

- Uniformizează modul de scriere a codului
- Facilitează citirea și 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 îmbunătățit și 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 indecșii, variabilele temporare și variabilele inițializate cu aceeași valoare

```
int Zone;  
int i, j, counter;  
int Mode = k = 0;
```
- Variabilele locale ar trebui inițializate înainte de utilizare
- Se va evita utilizarea variabilelor 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 dorește înlocuirea acestora (constante, fișiere, funcții)
- Toate fișierele header vor avea extensia “.h”
- Toate constantele simbolice definite cu #define vor fi scrise cu litere mari
- Numele unui fișier nu trebuie să depășească 14 caractere în lungime (cu extensie)

# Convenții de programare

---

- Variabilele locale, definițiile de tip (typedef), numele de fișiere cat și membrii unei structuri vor fi scriși cu litere mici
- Numele variabilelor și funcțiilor trebuie să fie semnificative și 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
  - Fișiere: initialData.txt, entry.txt, setFuncțiuni.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 Funcții  
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 fișier vor fi definite înainte de prima funcție din fișier
- Dacă se definesc constantele TRUE și 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 definiție (structura, enumerare, tip, etc) utilizată în mai multe fișiere va fi inclusă într-un fișier 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 [] și 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 câte ori este posibil

$x = f(a, b, c)$  și  $x = f(a, b, c, x)$   
sunt mai ușor 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* și 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 depășească un anumit grad de complexitate și 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; dacă vor fi utilizate trebuie îndeplinite următoarele cerințe:
  - Toate variabilele globale pentru un proiect vor fi definite într-un singur fișier
  - 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

---

- Fișierele antet vor defini o constantă simbolică pentru a permite includerea multiplă. Dacă fișierul se numește file.h constanta se poate numi FILE\_H

```
"example.h"  
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
...  
#endif
```

- Se recomandă ca fișierele antet să nu includă alte fișiere antet
- Pentru includerea unui fișier antet definit de utilizator se vor utiliza “ ”, iar pentru o bibliotecă standard < >

# Convenții de programare: Macrouri

---

- In macrourele tip funcție parametrii vor fi scriși 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));
```

- Macrourele complexe vor fi comentate
- Un macrou nu va depăși 10 linii



# Convenții de programare: Stil de programare

---

- Se va scrie o singură instrucțiune pe linie
- Tipul unei funcții și numele acesteia vor fi pe aceeași 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:

- după 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) și operanzii lor
- Înainte sau după operatorii primari ( "()", "[]", ".", "->")

# Stil de programare - Spațierea

---

## 2. Nu vor fi spații albe:

- Între caracterul # și directiva de preprocesare
  - **Nerecomandat**
    - #
    - define TEST 0
    - sau
    - # define TEST 0
  - **Recomandat**
    - #define TEST 0
- Între numele unei funcții și paranteza care îi urmează
- Între primul argument al funcției și paranteza deschisă
- Între ultimul argument al funcției și paranteza închisă
  - **Nerecomandat**
    - just\_return ( arg1, arg2 );
  - **Recomandat**
    - just\_return(arg1, arg2);
- Între parantezele deschisă, respectiv închisă și expresia unei instrucțiuni condiționale
  - if (x == y)...

# Stil de programare - Spațierea

---

## 3. Un singur spațiu

- va exista între expresia condițională a unei instrucțiuni și instrucțiune
  - **Nerecomandat**  
if(x == y)
  - **Recomandat**  
if (x == y)
- precede și urmează operatorii de atribuire, operatorii relaționali, operatorii logici, operatorii aritmetici (excepție cei unari) operatorii pe biți și 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 funcțiun just_return.
    */
    return(0);
}
```

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

# Utilizarea acoladelor și 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 aceeași 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 același nivel de includere vor fi indentate la aceeași coloană

```
if (condițion1 == 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 și vor fi indentate la același nivel cu codul pe care îl 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 același 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 aceeași 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 comentariu 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
- Instrucțiunile condiționale sau buclele complexe (mai mult de 10 linii necomentate) vor avea atașat un comentariu la acolada de închidere ce va indica închiderea instrucțiunii și unul la începutul sau în interiorul blocului ce va indica scopul acestuia

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

```
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.)