Lucrarea de laborator Nr. 01

Apeluri sistem. Funcţii de bibliotecă. Unelte de programare-compilatoare ”C” în UNIX. Tratarea erorilor. Procesarea liniei de comandă.

Aplicații demonstrative:

1. Utilizare compilator gcc.
   a. Primul program.
   b. Primul program concurent cu fire de execuție
   c. Primul program concurent cu procese
   d. Primul program paralel cu fire de execuție
2. Modalități de tratare a erorilor
3. Analizarea directă a argumentelor introduse pe linia de comadă.
4. Analizarea argumentelor introduse pe linia de comandă prin apel funcție *getopt()*.

Apeluri sistem. Funcții de bibliotecă.

Un apel de sistem

este metoda prin care un program solicită un serviciu de la nucleul sistemului de operare. Un apel sistem constă în fapt în apelul unei funcții sistem. Funcțiile sistem sunt declarate în fișiere cu extensia *.h* numite *fișiere header* (*.h* vine de la *header*), iar apelul lor *va* fi obiectul de studiu în lucrările de laborator care urmează.

Funcțiile sistem sunt declarate în fișierele *fișiere header*:

```
#include <unistd.h>
#include <sys/nume functie/.h>
...și altele...
```

Exemple de funcții sistem

fork()- crează un proces copil,
execl(), execle(), execlp(), execv(), execve(), execvp()- execută un fișier,
wait(), waitpid(), waitid()- așteaptă ca un proces să-și schimbe starea,
pipe()- crează un pipe anonim,
mkfifo()- crează FIFO (pipe cu nume),
send(), sendto(), sendmsg()- trimite un mesaj într-un socket,
recv, recvfrom, recvmsg- citește un mesaj dintr-un socket,
r**ead()-** citește de la un descriptor de fișier (file descriptor),
write()- scrie într-un descriptor de fișier (file descriptor),
getpid(), getppid()- obține identificarea unui proces, etc.

Funcțiile de bibliotecă

Funcțiile de bibliotecă sunt declarate în fișiere cu extensia *.h* numite *fișiere header* și ele completează facilitățile limbajului C cu posibilitatea de a prelucra direct șirurile de caractere, mulțimile, listele, tablourile precum și operații de intrare/ieșire.

Funcțiile de biblitecă sunt declarate în fișierele *fișiere header*:

```
#include < stdio.h>
#include <stdlib.h >
#include <math.h>
#include <string.h>
...și altele...
```

Exemple de funcții de bibliotecă

getchar()- returnează un caracter citit de la tastatură,
putchar()- afișează un caracter pe ecran; returnează caracterul afișat,
printf()- afișează pe ecran valorile dintr-o listă de argumente, conform formatului specificat,
sprintf()- conversie **date formatate spre string,**
getch()- citește un caracter de la tastatură fără să-l afișeze pe ecran,
getche()- citește un caracter de la tastatură și-l afișează pe ecran,

acos()- arccosinus, **asin()-** arcsinus, **atan()-** arctangentă, **atan2()-** arctangentă de y/x,
atoi()- conversie ascii la întreg,
atof()- conversie ascii la flotant, etc.

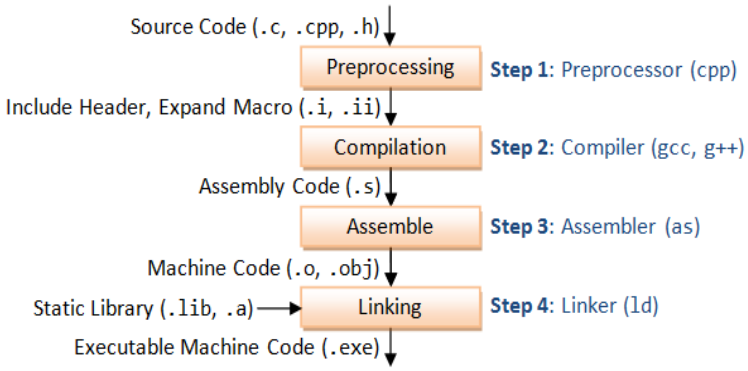
Unelte de programare-compilatoare ”C” în UNIX

gcc este compilatoul C din lumea UNIX-ului. El este 100% compatibil cu specificațiile ANSI pentru limbajul C, lucru valabil pentru platformele Solaris, Linux şi Microsoft Windows. Această compatibilitate ANSI 100% face ca aplicațiile, prin mici modificări, să poată fi portate relativ uşor între diferite platforme. **“C”** este limbajul în care a fost scris nucleul (kernel) al sistemului de operare Unix (Solaris, Linux, Aix, Sinix, etc) precum şi multe alte aplicații, incluzând o serie de aplicații din suita GNU, de exemplu *gnome-commander* similar cu *totalcmd.exe* din Windows sau *gedit* similar cu *notepad.exe* din Windows. Extensia validă C (compilator gcc) pentru fișierele ce conțin cod sursă este ‘.c’

g++ este compilatorul C++ care aduce în plus în lumea UNIX conceptul de structură de dată orientată obiect şi conceptul de metodă. Extensiile valide C++ (compilator g++) ale fişierelor ce conțin cod sursă sunt: ‘.cc’, ‘.cpp’, ‘.cxx’ sau ‘.C’

Modelul compilatorului C

Preprocesorul
Acceptă la intrare codul sursă C şi este responsabil pentru următoarele:
- elimină comentariile
- interpretează directivele speciale prefixate de simbolul #



Compilatorul de C
Primeşte la intrare fişierele sursă C, deja analizate şi prelucrate de către preprocessor şi va genera codul în limbaj de asamblare.
Assembler-ul
Crează codul obiect pornind de la sursele deja convertite în limbaj de asamblare. Va genera fişiere cu extensia .o (în WINDOWS sunt fişiere .obj).
Link-editorul
Dacă programul este realizat din mai multe fişiere sursă C, iar aceste fişiere folosesc variabile externe, link-editorul este componenta care va realiza legătura între definiţia variabilei (în unul din fişiere) şi utilizarea ei. Tot în faza de link-editare se face legătura între apelul funcţiei şi biblioteca unde apare definiţia ei, sau între apelul funcţiei şi definiţia ei, definiţie ce poate apărea într-un alt fişier al proiectului.

Tratarea erorilor

Tratarea erorilor este extrem de importantă în programarea UNIX. Apelurile de sistem pot eșua dintr-o varietate de motive care nu pot fi ignorate de programator. De exemplu, un apel de sistem care deschide un fișier poate eșua dacă fișierul nu este prezent sau dacă accesul la fișier a fost blocat de un alt program. Există, practic, două moduri diferite de a include procesarea erorilor.

1. Prima modalitate, prin apel funcție `perror()`, este ilustrată de segmentul de cod prezentat mai jos .

```
...

int fd;
...
if ((fd = open("fis", O_RDONLY))==-1)
        perror("nu pot sa deschid fisierul fis");
...
```

În acest segment de program, `open()` returnează `-1` atunci când eșuează. Eșecul apelului este testat și prin apelul funcției `perror()` care afișează mesajul `"nu pot sa deschid fisierul fis de eroare"` urmat de mesajul de eroare generat de apelul de sistem `open()` eșuat.

2. O a doua modalitate, prin apel funcție **`strerror(errno)`**, formă mai sofisticată de procesare a erorilor este arătată în următorul segment de cod.

```
...
#include<string.h>
#include<errno.h>
...
int fd;
...
if (( fd = open("fis", O_RDONLY))==-1)
        fprintf(stderr, "nu pot sa deschid fisierul fis %s\n",
        strerror(errno));
...
```

În exemplul de mai sus, eșecul apelului de sistem `open()` poziționează o variabilă globală `errno` la o anumită valoare ce reprezintă codul erorii. Semnificația codului de eroare este definită (prin intermediul unei macro) în `errno.h`. Apelul sistem `strerror(errno)` returnează un șir de caractere care indică natura precisă a erorii.

Procesarea liniei de comandă

O linie de comandă UNIX este formată din argumente, care sunt separate prin *blanc*, *tab*, sau ”\". Fiecare argument de pe linia de comandă este un *string* (șir de caractere terminat cu '\0'). Când un utilizator introduce o comandă corespunzătoare unui program executabil ”C”, *shell*-ul analizează linia de comandă și pasează programului rezultatul sub forma unui tablou de argumente (vector) de tip *string* definit prin declarația

```
char **argv
    sau
char *argv[]
```

Într-un caz concret considerăm că am startat programul *progr* cu următoarea linie de comandă:

```
$ ./progr arg1a arg2bb arg3ccc
```

Main-ul programului ”C” primește linia de comandă prin declarația:

```
int main(int argc, char **argv)
                SAU
int main(int argc, char *argv[])
```

`argv`, creat și transmis de *shell* programului, este un pointer către un tablou de pointeri (vector) de argumente terminat cu *NULL*. Fiecare element din vector este un pointer la rândul său către primul caracter al unui tablou unidimensional de caractere, care reprezintă valoarea argumentului de pe linia de comandă. Fiecare tablou unidimensional de caractere se termină cu '\0'. În fapt este vorba de un *string* (șir de caractere terminat implicit cu '\0').

Pentru cazul nostru concret *shell*-ul transmite programului un *argv* care punctează structura de mai jos:

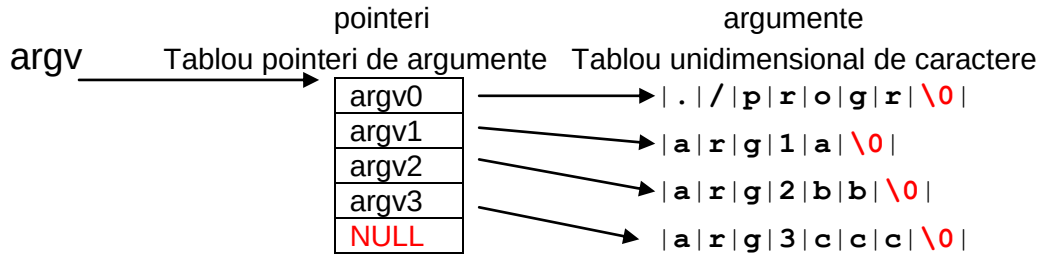


Figura de mai sus este corepunzătoare următoarei secvențe C:

```
char *argv []= {
                arg0,
                arg1,
                arg2,
                arg3,
                NULL
                };
```

```
char arg0[] = {'.','p','r', 'o', 'g' , 'r' ,'\0'};
char arg1[] = {'a','r','g', '1', 'a' , '\0'};
char arg2[] = {'a','r','g', '2', 'b' , 'b', '\0'};
char arg3[] = {'a','r','g', '3', 'c' , 'c', 'c', '\0'};
```

NOTĂ: argumentele se pot considera *string*-uri - șir de caractere terminat implicit cu '\0' -

```
char arg0[] = ".progr";
char arg1[] = "arg1a";
char arg2[] = "arg2bb";
char arg3[] = "arg3ccc";
```

argc, creat și transmis de *shell*, este un întreg care conține numărul de argumente de pe linia de comandă. În cazul de mai sus numărul de argumente de pe linia de comandă este egal cu 4.

Secvență de principiu prin care un program C poate citi direct argumentele liniei de c-dă este următoarea

```
...
int main(int argc, char *argv[])          // sau int main(int argc, char **argv)
...
 int i=0;
...
 printf("\nNumar argumente = %d\n", argc);
for(i=0;i<argc;i++)
{
 printf("\targv[%d] = %s\n",  i, argv[i]);
}
...
```

Argumentele dintr-o linie de comandă pot conține orice șiruri de caractere și este sarcina programului să le interpreteze așa cum crede de cuviință. O convenție în majoritatea comenzilor (dar nu toate) ale sistemului de operare Unix este de a plasa diferite opțiuni (switch) imediat după comandă. Opțiunile sunt litere singulare care sunt prefixate cu o cratimă (liniuță/minus) și pot fi urmate de argumente ale opțiunii într-o sintaxă flexibilă în care se pot introduce mai multe opțiuni argumente ale opțiunii împreună în orice ordine.
De exemplu, considerăm o linie de comandă de referință de forma:

```
$ ./gopt -a -c -ac -b BBBB -d DDDD
```

unde

- *-a, -c, -ac, -b, -d sunt opțiuni*
- *BBBB și DDDD sunt argumente atașate opțiunilor -b și respectiv -d*

Prin apelul funcției `getopt()` opțiunile din `argv` pot fi pot interpretate foarte ușor.
Pentru linia de comandă de referință, secvența C de mai jos va analiza opțiunile introduse:

```
...
int main(int argc, char *argv[])
. . .
while((optiune = getopt(argc, argv, "ab:cd:")) != -1)   //--- optiune= la fiecare apel va lua succesiv
{                                                       //                  valorile a, b, c și d
                                                        //        b:, d:  semnifică că opțiunile sunt urmate
                                                        //        de un argument
```

. . . se execută ceva în funcție de valoare ***opțiune*** respectiv ***argument* ...**

```
}
```

APLICAȚII DEMONSTRATIVE

1. Utilizare compilator gcc.

a) Primul program

Pe prompterul Unix din fereastra Terminal se startează compilatorul *gcc* indicându-se numele fişierului în care se găseşte codul sursa C şi opţional numele fişierului în care va rezulta codul executabil (valoarea implicită este *a.out*).

- Într-o fereastră terminal, se creaza cu un editor de texte simplu (ex. gedit) fișierul *salut.c* cu următorul conţinut, ce reprezintă un program foarte simplu în C:

```
/** Fisier salut.c  **/
/** Starteaza de 3 ori o functie  **/
/** Fiecare functie forteaza o intarziere de 1,2, sau 3 sec. **/
/**
   Timpul de real de executie a programului e in jur de 6 secunde
   indiferent de faptul ca calculatorul are mai multe procesoare
   sau procesorul are mai multe nuclee
**/
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void* helsl(void *argument) {
    int *i = (int *)(argument);
   int j = *i;
   fprintf(stderr, "Salut pe toata lumea %d - secunda/e\n", j);
   sleep (j);  // i =1,2 sau 3 secunde la fiecare trecere
}

int main(int argc, char* argv[])
{
   int i;
   int j[3]= {1,2,3}; // valoare sleep pt. fiecare functie
   for (i=0; i <3; i++)
   { // startez de 3 ori functia
     helsl(&j[i]);
    }

   fprintf(stderr, "\n");
    exit(123);    // cod retur pozitionat la 123
}
```

Operare

```
$ gcc -o salut salut.c          <--- compilare/link-editare program
$./salut                        <--- startare program
Salut pe toata lumea 1 - secunda/e
Salut pe toata lumea 2 - secunda/e
Salut pe toata lumea 3 - secunda/e

$ echo $?                       <--- (123 --afișare cod retur )
123

$ time ./salut                  <--- startare prin comanda time pentru a afla real time
Salut pe toata lumea 1 - secunda/e
Salut pe toata lumea 2 - secunda/e
Salut pe toata lumea 3 - secunda/e


real    0m6.003s                        <--- real time cca 6 sec.
user    0m0.001s
sys 0m0.002s
```

b) Primul program concurent cu fire de execuție

```
/** Fisier salutt.c  **/
/** Starteaza de 3 ori un thread  **/
/** Fiecare thread forteaza o intarziere de 1,2, sau 3 sec. **/
/**
   Daca calculatorul are ub procesor cu mai multe nuclee
   timpul real de executie a programului e in jur de 3 secunde.
   In cazul ca procesorul nu are mai multe nuclee atunci timpul
   real de executie a programului este in jur de 6 secunde
**/
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* helsl(void *argument) {
    int *i = (int *)(argument);
        int j = *i;
   fprintf(stderr, "Salut pe toata lumea %d - secunda/e\n", j);
   sleep (j);  // i =1,2 sau 3 secunde la fiecare trecere

}

int main(int argc, char* argv[])
{
   int i;
   pthread_t th[3];  // identificatori thread
   int j[3]= {1,2,3}; // valoare sleep pt. fiecare thread

   for (i=0; i <3; i++)
   { // startez de 3 ori thread-ul

     pthread_create(&th[i], NULL, helsl, &j[i]);

    }

   fprintf(stderr, "\n");
    pthread_exit(NULL);   // astept terminarea thread-urilor
}
```

Operare

```
$ gcc -o salutt salutt.c -lpthread
$ time ./salutt

Salut pe toata lumea 1 - secunda/e
Salut pe toata lumea 2 - secunda/e
Salut pe toata lumea 3 - secunda/e

real    0m3.004s                    <--- real time cca 3 sec.
user    0m0.000s
sys 0m0.004s
```

c) Primul program concurent cu procese

```
/** Fisier salutp.c  **/
/** Creaza si Starteaza de 3 procese in pieptene  **/
/** Fiecare proces forteaza o intarziere de 1,2, sau 3 sec. **/
/**
   Daca calculatorul are mai multe procesoare sau
   un procesor cu mai multe nuclee
   timpul real de executie a programului e in jur de 3 secunde.
   In cazul ca procesorul nu are mai multe nuclee atunci timpul
   real de executie a programului este in jur de 6 secunde
**/
#include <stdio.h>
#include <stdlib.h>
#include<sys/wait.h>
#include <unistd.h>
#include <pthread.h>

void* helsl(void *argument) {
    int *i = (int *)(argument);
        int j = *i;
   fprintf(stderr, "Salut pe toata lumea %d - secunda/e\n", j);
   //fprintf(stderr, "PID= %d  PPID=%d\n", getpid(), getppid());  // pune in evidenta ca s-au creat 3
procese in piptene
   sleep (j);  // i =1,2 sau 3 secunde la fiecare trecere

}

int main(int argc, char* argv[])
{
   int i, status;
   pid_t childpid;  // identificatori proces (pid)
   int j[3]= {1,2,3}; // valoare sleep pt. fiecare proces
   // fprintf(stderr, "PID= %d  PPID=%d\n", getpid(), getppid()); // pune in evidenta ca s-au creat 3
procese in pieptene
   for (i=0; i <3; i++)
   { // creez si startez trei procese copil
    childpid = fork();     // creare proces
    if (childpid == 0) {/* proces copil */ helsl(&j[i]); exit(0);}
    else
     {/* proces parinte */ /* reia for */; }
   }
   //fprintf(stderr, "\n");
   while (wait (&status)!= -1);   //  asteapta terminarea proceselor copil.

   exit(0);
}
```

Operare

```
$ gcc -o salutp salutp.c
$ time ./salutp
Salut pe toata lumea 1 - secunda/e
Salut pe toata lumea 3 - secunda/e
Salut pe toata lumea 2 - secunda/e

real    0m3.003s                    <--- real time cca 3 sec.
user    0m0.002s
sys 0m0.002s
```

d) Primul program paralel cu fire de execuție

```
/** saluttp.c **/
/*
  Starteaza de 3 ori in paralel o secventa de cod (o functie) sub forma unui thread.
  Fiecare thread startat in paralel forteaza o intarziere de 1,2, sau 3 sec.
*/
/*
    Daca calculatorul are un procesor cu mai multe nuclee
    sau mai multe procesoare, atunci timpul real de executie a programului este in jur de 3 secunde.
    In cazul in care calculatorul are un singur procesorul si acesta  nu are mai multe nuclee
    atunci timpul real de executie a programului este in jur de 6 secunde
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<pthread.h>   // pentru pthread_self()
#include <omp.h>       // pentru #pragma

/* functie startata in paralel */
void* helsl(void *argument) {
     int *i = (int *)(argument);
     int j = *i;
    fprintf(stderr, "Salut pe toata lumea %d - secunda/e\n", j);
    //fprintf(stderr, "TID= %u PID= %d  PPID=%d\n",
    //       pthread_self(), getpid(), getppid());  // pune in evidenta ca s-au creat 3 thread-uri
    sleep (j);  // i =1,2 sau 3 secunde la fiecare trecere
}

int main()
{
 int j[3]= {1,2,3}; // valoare sleep pt. fiecare thread paralel
 int i;
 #pragma omp parallel num_threads(3)    // [1] -- se va comenta si vizualiza efectele
 { /* codul din aceasta sectiune ruleaza in paralel de 3 ori */
   i= omp_get_thread_num();     // numar startare paralela = 0,1 sau 2
   helsl(&j[i]);
 } // fin cod in paralel

 fprintf(stderr, "main: Salut pe toata lumea %d secunda/e\n", j[i]);
 exit(0);
}
```

Operare

```
$ gcc -o saluttp saluttp.c -fopenmp -lpthread
time ./saluttp
Salut pe toata lumea 2 - secunda/e
Salut pe toata lumea 3 - secunda/e
Salut pe toata lumea 1 - secunda/e
main: Salut pe toata lumea 1 secunda/e
```

```
real    0m3.004s          <--- real time cca 3 sec.
user    0m0.000s
sys 0m0.004s


// se comenteaza linia [1] din codul sursa
$ gcc -o saluttp saluttp.c -fopenmp -lpthread
$ time ./saluttp
Salut pe toata lumea 1 - secunda/e
main: Salut pe toata lumea 1 secunda/e

real    0m1.003s
user    0m0.001s
sys 0m0.003s
```

2. Modalități de tratare a erorilor

```
// Fisier: mtrater.c
// Incercare de deschidere a unui fisier inexistent //
//  + Doua moduri de tratare a erorilor

#include<string.h>        // pt. strerror())
#include<stdio.h>         // pt. perror() si fprintf()
#include<stdlib.h>        // pt. exit()
#include<fcntl.h>         // pt. open())
#include<errno.h>          // pt. errno din strerror())

int main(int argc, char *argv[])          // sau int main(int argc, char **argv)
{
  int fd;
  /* Incercare de deschidere a unui fisier inexistent */
  if ((fd = open("fis", O_RDONLY))==-1)  /* Modul 1 de tratare a erorilor prin apel perror() */
    perror("M1 cu perror()-Nu pot sa deschid fisierul fis din urmatorul motiv");

  if (( fd = open("fis", O_RDONLY))==-1) /* Modul 2 de tratare a erorilor prin apel strerror() */
    fprintf(stderr, "\nM2 cu strerror()-Nu pot sa deschid fisierul fis din urmatorul motiv: %s\n",
    strerror(errno));
 exit(123);     // cod retur
};
```

Operare

```
$ gcc -o mtrater mtrater.c
$ ./mtrater
M1 cu perror()-Nu pot sa deschid fisierul fis din urmatorul motiv: No such file or directory

M2 cu strerror()-Nu pot sa deschid fisierul fis din urmatorul motiv: No such file or directory

$ echo $?
123
$
```

3. Analizarea directă a argumentelor introduse pe linia de comadă

```
// Fisier: progr.c
//  Citirea/Afisarea argumentelor de pe linia de comanda

#include <stdio.h>
```

```
int main(int argc, char *argv[])          // sau int main(int argc, char **argv)
{
 int i;
 printf("\nNumar argumente = %d\n", argc);
 printf("\nAfisare argumente\n");
 for (i=0; i<argc; i++)
 {            // afisare argumente
  printf("\targument[%d] = %s\n", i, argv[i]);
 };
 return 0;
};
```

Operare

```
$ gcc -o progr progr.c
$ ./progr arg1a arg2bb arg3ccc

Numar argumente = 4

Asisare argumente
   argument[0] = ./progr
   argument[1] = arg1a
   argument[2] = arg2bb
   argument[3] = arg3ccc
```

4. Analizarea argumentelor introduse pe linia de comandă prin apel funcție *getopt()*

Exemplul de mai jos ilustreaza o utilizare simplă a functiei `getopt()` prin care se analizează mai multe elemente introduse pe linia de comandă.

- Un element de pe linia de comadă va fi o literă prefixată de o cratimă, urmată în unele cazuri de un argument opțional.
- Apelul funcției `getopt()` din program definește următoarele opțiuni care pot fi introduse în linia de comandă "`ab:cd:`", adică `-a`, `-d` urmat de un `argument` al opțiunii, `-c` și `-d` urmat de un a`rgument` al opțiunii.
- Tratarea opțiunii se face într-o structură de `switch/case`, iar opțiunile sunt selectate pe rând până la epuizare într-o structura `while`.
- Variabila `optarg` este setatată de funcția `getopt()` și conține `argumentul` atașat opțiunii, pentru opțiunile ce pretind acest lucru.

```
/**  Fisier: gopt.c   **/
/*  Functia getopt() pentru a analiza mai multe
    optiuni si argumente de pe linia de comanda
*/
#include <stdio.h>    /* pentru functiile printf(), fprintf() */
#include <stdlib.h>     /* pentru functia exit() */
#include <unistd.h>     /* pentru functia getopt(), optarg */
int main(int argc, char *argv[])
{
   int optiune;
   int i;
   printf("argc = %d\n", argc);
   if (argc == 1) { // Un prim control al linei de comanda.
      fprintf(stderr, "Eroare: lipsa optiuni pe linia de comanda\n");
      exit(1);
   }
   /* Afisare argv */
   for (i=0; i<argc; i++)
      printf("arg[%d] = \"%s\"\n", i, argv[i]);
   /* Interpretare optiuni argv */
```

```
while((optiune = getopt(argc, argv, "ab:cd:")) != -1)   // un al doilea control al linei de comanda
                    // a = optiune -a , c = optiune -c
                    // b: = optiune -b urmata de un argument pozitionat in optarg
                    // d: = optiune -d urmata de un argument pozitionat in optarg
{
   printf("Optiuniunea citita de pe linia de comanda este: %c \n", optiune);
   switch(optiune)
   {
     case 'a' : // NU primeste un sir de caractere dupa optiunea -a
       //se executa ceva
       printf("\tPrelucrez optiune %c \n", optiune);
       break;
     case 'b' : // primeste un sir de caractere dupa optiunea -b
      //se executa ceva
       printf("\tPrelucrez optiune %c si argumentul %s \n", optiune, optarg);
       break;
     case 'c' : // NU primeste un sir de caractere dupa optiunea -c
      //se executa ceva
       printf("\tPrelucrez optiune %c \n", optiune);
       break;
     case 'd' : // primeste un sir de caractere dupa optiunea -d
      //se executa ceva
       printf("\tPrelucrez optiune %c si argumentul %s \n", optiune, optarg);
       break;
     case  '?' : // optiune nerecunoscuta
       fprintf(stderr, "Utilizare: %s cu optiuni sau argumente eronate\n",  argv[0]);
       exit(EXIT_FAILURE);
    }

   }
 exit(0);
}
```

Operare

$gcc -o gopt gopt.c

1. Startare cu toate opțiunile și argumentele opționale prevăzute.
   Atenție: Se remarcă dublarea prelucrării care ar trebui evitată prin program.
   (vezi prelucrare optiune -a care a fost dată de două ori de exemplu)

```
$ ./gopt -a -c -ac -b BBBB -d DDDD
argc = 8
arg[0] = "./gopt"
arg[1] = "-a"
arg[2] = "-c"
arg[3] = "-ac"
arg[4] = "-b"
arg[5] = "BBBB"
arg[6] = "-d"
arg[7] = "DDDD"
Optiuniunea citita de pe linia de comanda este: a
   Prelucrez optiune a
Optiuniunea citita de pe linia de comanda este: c
   Prelucrez optiune c
Optiuniunea citita de pe linia de comanda este: a
   Prelucrez optiune a
Optiuniunea citita de pe linia de comanda este: c
   Prelucrez optiune c
Optiuniunea citita de pe linia de comanda este: b
   Prelucrez optiune b si argumentul BBBB
Optiuniunea citita de pe linia de comanda este: d
```

```
   Prelucrez optiune d si argumentul DDDD
```

2. Startare cu o singură opțiune

```
$ ./gopt -a
argc = 2
arg[0] = "./gopt"
arg[1] = "-a"
Optiuniunea citita de pe linia de comanda este: a
   Prelucrez optiune a
```

3. Startare cu o singură opțiune care pretinde un argument opțional--: EROARE APEL

```
$ ./gopt -b
argc = 2
arg[0] = "./gopt"
arg[1] = "-b"
./gopt: option requires an argument -- 'b'
Optiuniunea citita de pe linia de comanda este: ?
Utilizare: ./gopt cu optiuni sau argumente eronate
```

4. Startare cu două opțiuni puse împreună.

```
Opțiunea -a nu pretinde argument opțional.
Opțiunea -d  pretinde un argument opțional.

$ ./gopt -ad DDDD
argc = 3
arg[0] = "./gopt"
arg[1] = "-ad"
arg[2] = "DDDD"
Optiuniunea citita de pe linia de comanda este: a
   Prelucrez optiune a
Optiuniunea citita de pe linia de comanda este: d
   Prelucrez optiune d si argumentul DDDD
$
```