Parametrii liniei de comandă. Preprocesorul. Funcţii cu număr variabil de parametri.

Resposabili:Responsabili:

- Dorinel Filip (CA 2016-2020) [mailto:ion_dorinel.filip@cti.pub.ro]
- Darius Neațu (CA 2016-2020) [mailto:neatudarius@gmail.com]
- Mihaela Vasile (2015) [mailto:mihaela.vasile@gmail.com]

Ultima modificare: 09.12.2018

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

- să interpreteze şi să manipuleze parametrii liniei de comandă
- să scrie programe care comunică cu exteriorul (utilizatorul, sistemul de operare, alte programe) prin intermediul parametrilor liniei de comandă
- să scrie şi să folosească macrodefiniţii
- să utilizeze macrodefiniţiile în lucrul cu mai multe fişiere
- să evite erorile care apar frecvent în lucrul cu macrodefiniţii
- să realizeze funcţii care primesc un număr variabil de parametri
- să folosească macroinstrucţiunile necesare manipulării stivei de parametri a unei funcţii cu număr variabil de parametri

- Laboratorul curent introduce mai multe notiuni. Pe multe dintre acestea le veti mai intalni si la alte cursuri precum USO si SO.
- Sectiunile marcate cu **studiu de caz** nu sunt obligatorii, dar recomandam parcurgerea intregului material pentru o viziune de ansamblu mai buna.

Parametrii liniei de comandă

Pentru a controla execuţia unui program, de multe ori este de dorit furnizarea datelor de lucru înaintea lansării în execuţie a programului, acesta urmând să se execute apoi fără intervenţia utilizatorului (aşa-numitul „batch mode”). Acest lucru se realizează prin intermediul parametrilor liniei de comandă. (Un exemplu cunoscut este lansarea compilatorului gcc în linia de comandă cu diverse argumente, care îi spun ce şi cum sa compileze.)

Din punct de vedere al utilizatorului, parametrii liniei de comandă sunt simple argumente care se adaugă după numele unui program, în linia de comandă, la rularea sa. Elementele acestei liste de argumente sunt şiruri de caractere separate de spaţii. Argumentele care conţin spaţii pot fi combinate într-un singur argument prin inchiderea acestuia între ghilimele. Shell-ul este cel care se ocupă de parsarea liniei de comandă şi de crearea listei de argumente.

Exemplu de apelare a unui program cu argumente în linia de comandă:

```
gcc -Wall -I/usr/include/sys -DDEBUG -o "My Shell" myshell.c
```

În acest caz, argumentele liniei de comandă sunt în acest caz:

- gcc
- -Wall

- -I/usr/include/sys
- -DDEBUG
- -o
- My Shell
- myshell.c

Din punct de vedere al programatorului, parametrii liniei de comandă sunt accesibili prin utilizarea parametrilor funcţiei main(). Astfel, când se doreşte folosirea argumentelor liniei de comandă, funcţia main() se va defini astfel:

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

Astfel, funcţia **main()** primeşte, în mod formal, doi parametri, un întreg şi un vector de şiruri de caractere. Numele celor două variabile nu e obligatoriu să fie **argc** şi **argv**, dar tipul lor, da. Semnificaţia lor este următoarea:

- **int argc** (argument count) - reprezintă numărul de parametrii ai liniei de comandă. După cum se vede din exemplul anterior, există cel puţin un parametru, acesta fiind chiar numele programului (numele care a fost folosit pentru a lansa în execuţie programul - şi care poate fi diferit de numele executabilului - de exemplu prin crearea unui symlink, în Linux).
- **char *argv[]** (arguments value) - reprezintă un vector de şiruri de caractere, având **argc** elemente (indexate de la **0** la **argc - 1**). Întotdeauna **argv[0]** conţine **numele programului**.

Parametrii liniei de comandă se pot accesa prin intermediul vectorului **argv** şi pot fi prelucraţi cu funcţiile standard de prelucrare a şirurilor de caractere.

Fie urmatorul cod sursa care afiseaza numarul de argumente primite, apoi afiseaza numele extras al executabilului din calea primitita ca argument.

```
#include <stdio.h>
#include <string.h> // strrchr

int main(int argc, char *argv[]) {
    // afiseaza numarul de argumente + primul argument
    printf("Am primit %d argument(e)\n", argc);
    printf("argv[0] = %s\n", argv[0]);

    // extrage numele executabilului
    char *program_name = strrchr(argv[0], '/') + 1;
    printf("Eu sunt programul %s\n", program_name);

    return 0;
}
```

Pentru compilare vom rula comanda:

```
gcc -Wall test.c -o gigel
```

Exemple de rulare:

```
darius@pc ~ $ ./gigel
Am primit 1 argument(e)
argv[0] = ./gigel
Eu sunt programul gigel

darius@pc ~ $ /home/darius/gigel
Am primit 1 argument(e)
argv[0] = /home/darius/gigel
Eu sunt programul gigel

darius@pc ~ $ ./gigel Gigel e pe val!
```

```
Am primit 5 argument(e)
argv[0] = ./gigel
Eu sunt programul gigel
```

Preprocesorul

Preprocesorul este componenta din cadrul compilatorului C care realizează preprocesarea. În urma acestui pas, toate instrucţiunile de preprocesare sunt **înlocuite (substituite)**, pentru a genera cod C „pur”. Preprocesarea este o **prelucrare exclusiv textuală** a fişierului sursă. În acest pas nu se fac nici un fel de verificări sintactice sau semantice asupra codului sursă, ci doar sunt efectuate substituţiile din text. Astfel, preprocesorul va prelucra şi fişiere fără nici un sens în C.

Se consideră următorul exemplu:

```
#define EA Ana
#define si C
#ifdef CIFRE
#define CINCI 5
#define DOUA 2
EA are mere. Mara are DOUA pere shi CINCI cirese.
#endif
Vasilica vrea sa cante o melodie in si bemol.
```

La rularea comenzii

```
gcc -E -DCIFRE rubbish.c
```

se va obţine următoare ieşire (se cere compilatorului să execute doar pasul de preprocesare (-E), definind în acelaşi timp şi simbolul CIFRE (-DCIFRE) :

```
# 1 "rubbish.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "rubbish.c"
Ana are mere. Mara are 2 pere shi 5 cirese.
Vasilica vrea sa cante o melodie in C bemol.
```

Cele mai importante instrucţiuni de preprocesare sunt prezentate în continuare.

Incluziune

Probabil cea mai des folosită instrucţiune de preprocesare este cea de incluziune, de forma

```
#include <nume_fişier>
```

sau

```
#include "nume_fisier"
```

care are ca rezultat înlocuirea sa cu conţinutul fişierului specificat de **nume_fişier**. Diferenţa dintre cele două versiuni este că cea cu paranteze unghiulare caută **nume_fişier** doar în directorul standard de fişiere antet (numit deobicei **include**), iar cea cu ghilimele caută atât în directorul **include** cât şi în directorul curent.

Definirea de simboluri

Definirea de simboluri este cel mai des folosită în conjuncţie cu instrucţiunile de procesare condiţionată, fiind folosită pentru activarea şi dezactivarea unor segmente de cod în funcţie de prezenţa unor simboluri. Definirea unui simbol se face în cod cu instrucţiunea

```
#define SIMBOL
```

sau se poate realiza şi la compilare, prin folosirea flagului -D al compilatorului (după cum am văzut în exemplul precedent).

Un simbol poate fi de asemenea „şters” folosind instrucţiunea

```
#undef SIMBOL
```

în cazul în care nu se mai doreşte prezenţa simbolului de preprocesor ulterior definirii sale.

Codul urmator defineste cateva simboluri. Simbolul CHANGE este definit la compilare. Definirea acestuia determina modificarea simbolului GIGEL.

```
#include <stdio.h>

// am definit GIGEL ca fiind 1
#define GIGEL 1

// Verific daca este definit simbolul CHANGE
#ifdef CHANGE
#undef GIGEL // sterg GIGEL
#define GIGEL 0 // redefinesc simbolul cu alta valoare
#endif

int main() {
    printf("GIGEL = %d\n", GIGEL);
    return 0;
}
```

Observati diferentele intre cele doua rulari.

```
darius@pc ~ gcc -Wall -Wextra test.c -o test -DCHANGE=2017
darius@pc ~ $ ./test
GIGEL = 0

darius@pc ~ $ gcc -Wall -Wextra test.c -o test
darius@pc ~ $ ./test
GIGEL = 1
```

Definirea de macro-uri

Instrucţiunile de preprocesare mai pot fi folosite şi pentru definirea de constante simbolice şi macroinstrucţiuni. De exemplu

```
#define IDENTIFICATOR valoare
```

va duce la înlocuirea peste tot în cadrul codului sursă a şirului **IDENTIFICATOR** cu şirul **valoare**. Înlocuirea nu se face totuşi în interiorul şirurilor de caractere.

Un exemplu foarte bun este definirea unor macro-uri pentru dimensiunile tablourilor alocate static.

```
#define NMAX 100
#define MMAX 10
#define LMAX 10000

...
int a[NMAX][MMAX];
...
int v[LMAX];
...
```

O macroinstrucţiune este similară unei constante simbolice, ca definire, dar acceptă parametrii. Este folosită în program în mod asemănător unei funcţii, dar la compilare, ea este înlocuită în mod textual cu corpul ei. În plus, nu se face nici un fel de verificare a tipurilor. Spre exemplu:

```
#define MAX(a, b) a > b ? a : b
```

va returna maximul dintre a şi b, iar

```
#define DUBLU(a) 2*a
```

va returna dublul lui a.

Deoarece preprocesarea este o prelucrare textuală a codului sursă, în cazul exemplului de mai sus, macroinstrucţiunea în forma prezentată nu va calcula întotdeauna dublul unui număr.

Astfel, la o folosire de forma:

```
DUBLU(a + 3)
```

în pasul de preprocesare se va genera expresia

```
2*a+3
```

care bineînţeles că nu realizează funcţia dorită.

Pentru a evita astfel de probleme, este bine ca întotdeauna în corpul unui macro, numele „parametrilor” să fie închise între paranteze:

```
#define SQUARE(a) (a)*(a)
```

Fie simbolul SQUARE, care inlocuieste o expresie cu expresia ridicata la patrat. Acesta poate fi definit astfel.

```
#define SQUARE(a) (a)*(a)
```

Este aceasta expresie este corecta? (oricare ar fi a, va calcula patratul lui a?) Ce se intampla daca inaintea lui **SQUARE** aplicam un operand care are prioritate mai mare decat ***** ?

```
#include <stdio.h>

#define SQUARE(a) (a) * (a)

int main() {
    // ceea ce se obtine
    printf("%d\n", ~SQUARE(2));

    // ceea ce se doreste
    printf("%d\n", ~4);

    return 0;
}
```

```
gcc -Wall -Wextra test.c  -o test
./test
-6 // ceea ce se obtine
-5 // ceea ce se doreste
```

Operatiile nu se vor executa in ordinea in care dorim. **~** are precedenta mai mare decat ***** , deci mai intai se va executa operatia pe biti apoi inmultirea.

Pentru a fi convinsi ca se intampla acest lucru, putem compila cu -E si sa inspectam codul urmator.

```
... // continut din stdio.h, alte simboluri etc.

int main() {
    printf("%d\n", ~(2) * (2));

    printf("%d\n", ~4);

    return 0;
}
```

Pentru a corecta aceasta greseala, putem adauga inca o paranteza in definitia lui SQUARE.

```
#define SQUARE(a) ( (a) * (a) )
```

Pentru noua sursa C obtinem urmatorul rezultat.

```
gcc -Wall -Wextra test.c  -o test
./test
-5 // ceea ce se obtine
-5 // ceea ce se doreste
```

Sa incercam sa definim un simbol corect pentru aflarea maximului intre 2 expresii numerice.

Mai sus a fost prezentata urmatoarea varianta.

```
#define MAX(a, b) a > b ? a : b
```

Fie urmatorul cod C.

```
#include <stdio.h>

#define MAX(a, b) a > b ? a : b

int main() {
    printf("max = %d\n", 1 << MAX(19 + 1, 10 + 1));
    return 0;
}
```

Ceea ce ne-am astepta este sa se evalueze cele doua expresii (19 + 1 == 20, 10 + 1 == 11), sa sa calculeze maximul (20) si sa se afiseze **1048576** (1 « 20).

```
gcc -Wall -Wextra test.c  -o test
./test
test.c: In function ‘main’:
test.c:6:28: warning: suggest parentheses around ‘+’ inside ‘<<’ [-Wparentheses]
     printf("max = %d\n", 1 << MAX(19 + 1, 10 + 1));
                            ^
max = 20
```

Din nou precedenta operatorilor **+** si **«** impiedica obtinerea rezulatului dorit.

Solutia este sa punem si la MAX paranteze.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Vom obtine urmatorul rezultat.

```
gcc -Wall -Wextra test.c  -o test
./test
max = 1048576
```

Sa se defineasca un simbol SWAP care sa interschimbe doua variabile de acelasi tip, tipul fiind precizat.

Pornim de la o posibila secventa de cod care realizeaza acest lucru pentru doua variabile int.

```
int a, b;
...
int tmp = a;
a = b;
b = tmp;
```

Observati ca substituind textual cuvantul “int” cu “double”, bucata de cod nu isi schimba logica. Generalizarea acestui cod, presupune generalizarea tipului acestor variabile. Daca am nota acest type cu “type” (presupunand ca exista acest tip), atunci codul obtinut este urmatorul.

```
type a, b;
...
type tmp = a;
a = b;
b = tmp;
```

Solutia intuitiva de definire a simbolului SWAP este urmatoarea.

```
#include <stdio.h>

#define SWAP(type,x,y) type tmp = x; x = y; y = tmp;

int main() {
    // interschimbare int
    int x = 1, y = 2;
    printf("before: x = %d y = %d\n", x, y);
    SWAP(int, x, y)
    printf("after : x = %d y = %d\n", x, y);

    return 0;
}
```

In urma precompilarii, functia main arata astfel.

```
...
int main() {

    int x = 1, y = 2;
    printf("before: x = %d y = %d\n", x, y);
    int tmp = x; x = y; y = tmp;
    printf("after : x = %d y = %d\n", x, y);

    return 0;
}
```

Sursa compileaza si ruleaza corect.

```
gcc -Wall -Wextra test.c  -o test
./test

before: x = 1 y = 2
after : x = 2 y = 1
```

Urmatoarea sursa nu va compila, intrucat la folosirea repetata a lui SWAP in acelasi scop, variabila tmp va fi declarate de mai multe ori.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SWAP(type,x,y) type tmp = x; x = y; y = tmp;

int main() {
    // interschimbare int
    int x = 1, y = 2;
```

```
    printf("before: x = %d y = %d\n", x, y);
    SWAP(int, x, y)
    printf("after : x = %d y = %d\n", x, y);


    // interschimbare pointeri (char *)
    char *p = strdup("A");
    char *q = strdup("B");
    printf("before: p = (%s) q = (%s)\n", p, q);
    SWAP(char *, p, q);
    printf("after : p = (%s) q = (%s)\n", p, q);

    // elibereaza memoria alocata dinamic
    free(p);
    free(q);

    return 0;
}
```

```
 gcc -Wall -Wextra test.c  -o test && ./test
test.c: In function ‘main’:
test.c:5:29: error: conflicting types for ‘tmp’
 #define SWAP(type,x,y) type tmp = x; x = y; y = tmp;
                             ^
test.c:19:5: note: in expansion of macro ‘SWAP’
     SWAP(char *, p, q);
     ^
test.c:5:29: note: previous definition of ‘tmp’ was here
 #define SWAP(type,x,y) type tmp = x; x = y; y = tmp;
                             ^
test.c:11:5: note: in expansion of macro ‘SWAP’
     SWAP(int, x, y)
```

Solutia este sa folosim un scop local pentru tmp (scope [https://en.wikipedia.org/wiki/Scope_(computer_science)#Levels_of_scope]). Pentru lizibilitate putem scrie instructiunile din SWAP pe mai multe randuri. In acest caz vom pune caracterul '\' pentru a marca faptul ca simbolul continua pe randul urmator.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SWAP(type,x,y) {\
    type tmp = x; \
    x = y; \
    y = tmp; \
}

int main() {
    // interschimbare int
    int x = 1, y = 2;
    printf("before: x = %d y = %d\n", x, y);
    SWAP(int, x, y)
    printf("after : x = %d y = %d\n", x, y);


    // interschimbare pointeri (char *)
    char *p = strdup("A");
    char *q = strdup("B");
    printf("before: p = (%s) q = (%s)\n", p, q);
    SWAP(char *, p, q);
    printf("after : p = (%s) q = (%s)\n", p, q);

    // elibereaza memoria alocata dinamic
    free(p);
    free(q);

    return 0;
}
```

Inspectati codul dupa etapa de preprocesare. Compilati si rulati codul.

- ATENTIE! O variabila este vizibila in scopul (scope [https://en.wikipedia.org/wiki/Scope_(computer_science)#Levels_of_scope]) in care a fost declarata. La iesirea din scop, zona de memorie corespunzatoare este eliberata(Scope and Lifetime of Variables in C [https://blog.feabhas.com/2010/09/scope-and-lifetime-of-variables-in-c/]).

Instrucţiuni de compilare condiţionată

Instrucţiunile de compilare condiţionată sunt folosite pentru a „ascunde” fragmente de cod în funcţie de anumite condiţii. Formatul este următorul:

```
#if conditie
....
#else
....
#endif
```

unde conditie este este o expresie constantă întreagă. Pentru realizarea de expresii cu mai multe opţiuni se poate folosi şi forma **#elif**:

```
#if conditie
...
#elif conditie2
...
#elif conditie3
...
#else
...
#endif
```

De obicei condiţia testează existenţa unui simbol. Scenariile tipice de folosire sunt:

- dezactivarea codului de debug o dată ce problemele au fost remediate
- compilare condiţionată în funcţie de platforma de rulare
- prevenirea includerii multiple a fişierelor antet

În aceste cazuri se foloseşte forma

```
#ifdef SIMBOL
```

sau

```
#ifndef SIMBOL
```

care testează dacă simbolul SIMBOL este definit, respectiv nu este definit.

Directivele de preprocesor ar putea fi utile pentru a scrie cod portabil. De exemplu, putem identifica tipul sistemului de operare sau arhitectura procesorului. In exemplul urmator, se doreste folosirea unei functii f care sa aiba implementare de dependenta de sistemul de operare.

```
#include <stdio.h>

#ifdef _WIN32 // is _WIN32 defined?
//do something for windows like #include <windows.h>
void f() {
    printf("_WIN32\n");
}
#elif defined __unix__ // or is __unix__ defined?
//do something for unix like #include <unistd.h>
void f() {
```

```
    printf("__unix__\n");
}
#elif defined __APPLE__ // or is __APPLE__ defined?
//do something for mac
void f() {
    printf("__APPLE__\n");
}
#endif

int main() {
    f();
    return 0;
}
```

Sa ne uitam la fisierul C obtinut in urma etapei de preprocesare pe un **sistem UNIX**.

```
gcc -Wall -Wextra -E test.c
```

```
...
void f() {
    printf("__unix__\n");
}
int main() {
    f();
    return 0;
}
```

Se observa disparitia acelor directive de preprocesare si faptul ca sursa obtinuta contine doar implementarea pentru Unix. Daca rulam aceeasi comanda pe Windows (32bit sau 64 bit) sursa obtinut va fi urmatoarea.

```
...
void f() {
    printf("_WIN32\n");
}
int main() {
    f();
    return 0;
}
```

La rularea executabilului, se va afisa pe ecran stringul corespunzator sistemului de operare pe care a fost compilat executabilul.

Includere multipla

Prevenirea includerii multiple a fişierelor antet se realizează astfel:

```
#ifndef _NUME_FISIER_ANTET_
#define _NUME_FISIER_ANTET_
  /* corpul fisierului antet */
  /* prototipuri de functii, declaratii de tipuri si de constante */
#endif
```

Astfel, la prima includere a fişierului antet, simbolul **_NUME_FISIER_ANTET_** nu este definit. Preprocesorul execută ramura **#ifndef** în care este definit simbolul **_NUME_FISIER_ANTET_** şi care conţine şi corpul - conţinutul util - al fişierului antet. La următoarele includeri ale fişierului antet simbolul **_NUME_FISIER_ANTET_** va fi definit iar preprocesorul va sări direct la sfârşitul fişierului antet, după **#endif**.

Vom face o ierarhie de fisiere pentru a vedea utilizarea **garzilor** (include guard [https://en.wikipedia.org/wiki/Include_guard]).

Fie sursa test.c in care dorim sa apelam functia **pc** din fisierul antet **pc.h** si functia **run** din fisierul antet **uso.h** .

```
#include <stdio.h>
#include "uso.h"
#include "pc.h"

int main() {
    pc();
    run();
    return 0;
}
```

Fisierul **pc.h** are urmatorul continut.

```
#include "uso.h"

void write_in_c() {
    printf("write code in C\n");
}

void pc() {
    write_in_c();
    compile();
    run();
}
```

Fisierul **uso.h** are urmatorul continut.

```
void compile() {
    printf("compile with gcc\n");
}

void run() {
    printf("run\n");
}
```

Daca compilam fisierul sursa C vom obtine urmatoarele erori.

```
gcc -Wall -Wextra test.c  -o test
In file included from pc.h:1:0,
                 from test.c:3:
uso.h:1:6: error: redefinition of ‘compile’
 void compile() {
      ^
In file included from test.c:2:0:
uso.h:1:6: note: previous definition of ‘compile’ was here
 void compile() {
      ^
In file included from pc.h:1:0,
                 from test.c:3:
uso.h:5:6: error: redefinition of ‘run’
 void run() {
      ^
In file included from test.c:2:0:
uso.h:5:6: note: previous definition of ‘run’ was here
 void run() {
```

Functiile compile si run au fost definite de doua ori. Pentru a inspecta aceasta problema oprim, din nou, compilarea in faza de preprocesare.

```
...
# 2 "test.c" 2
# 1 "uso.h" 1

# 1 "uso.h" // aici se include prima oara "uso.h" (linia ''#include "uso.h"'' din "test.c")
void compile() {
    printf("compile with gcc\n");
}

void run() {
    printf("run\n");
```

```
}
# 3 "test.c" 2
# 1 "pc.h" 1
# 1 "uso.h" 1 // aici se include a doua oara "uso.h" (linia ''#include "pc.h"'' din "test.c" include "pc.h",
              //care contine linia ''#include "uso.h"'', instructiune in urma careia se va include din nou "uso.h"

void compile() {
    printf("compile with gcc\n");
}

void run() {
    printf("run\n");
}
# 2 "pc.h" 2

void write_in_c() {
    printf("write code in C\n");
}

void pc() {
    write_in_c();
    compile();
    run();
}
# 4 "test.c" 2

int main() {
    pc();
    run();
    return 0;
}
```

Aplicam solutia de mai sus. Definim in fiecare fisier header cate un simbol. Definim simbolul **PC_H** in **pc.h**.

```
#ifndef PC_H // daca este prima oara
             // cand acest fiser este inclus
#define PC_H // defineste simbolul pentru a nu
             // nu permite o includere ulterioara

#include "uso.h"

void write_in_c() {
    printf("write code in C\n");
}

void pc() {
    write_in_c();
    compile();
    run();
}

#endif // sfarsitul codului care depinde de
       // existenta simbolului PC_H
```

Definim simbolul **USO_H** in fisierul **uso.h**.

```
#ifndef USO_H // daca este prima oara
              // cand acest fiser este inclus
#define USO_H // defineste simbolul pentru a nu
              // nu permite o includere ulterioara

void compile() {
    printf("compile with gcc\n");
}

void run() {
    printf("run\n");
}

#endif // sfarsitul codului care depinde de
       // existenta simbolului USO_H
```

Fisierele antet vor fi incluse o singura data, codul va compila.

```
gcc -Wall -Wextra test.c  -o test
./test

write code in C
compile with gcc
run
run
```

Alte instrucţiuni

```
#pragma expresie
```

Sunt folosite pentru a controla din codul sursă comportamentul compilatorului (modul în care generează cod, alinierea structurilor, etc.) iar formatul lor diferă de la compilator la compilator. Pentru a determina ce opţiuni **#pragma** aveţi la dispoziţie consultaţi manualul compilatorului.

```
#error MESSAGE
```

La întâlnirea acestei instrucţiuni de preprocesare compilatorul va raporta o eroare, având ca text explicativ mesajul **MESSAGE**.

```
#line NUMBER FILENAME
```

Această instrucţiune de preprocesare modifică numărul liniei curente în valoarea specificată de **NUMBER**. În cazul în care este prezent şi parametru opţional **FILENAME** este modificat şi numele fişierului sursă curent. Astfel, mesajele de eroare şi avertismentele produse de compilator vor folosi numere de linie (şi eventual nume de fişiere) inexistente, „imaginare”, conform acestei instrucţiuni.

Funcţii cu număr variabil de parametri

Introducere

Marea majoritate a funcţiilor discutate până acum şi totalitatea funcţiilor realizate în cadrul laboratorului până în acest moment primeau ca parametrii un număr prestabilit de parametrii, de tipuri bine precizate. Acest lucru se datorează faptului că limbajul C este un limbaj strong-typed [http://en.wikipedia.org/wiki/Strongly-typed_programming_language]. În cele mai multe cazuri acesta este un aspect pozitiv întrucât compilatorul va sesiza de la compilare situaţiile în care se încearcă pasarea unui număr eronat de parametrii sau a unor parametrii de tipuri necorespunzătoare.

Limbajul C permite totuşi şi declararea şi folosirea funcţiilor cu un număr (şi eventual tip) variabil de parametri. Astfel, numărul şi tipul tuturor parametrilor va fi cunoscut doar la rulare, biblioteca standard C punând la dispoziţie o serie de definiţii de tipuri şi macro definiţii care permit parcurgerea listei de parametri a unei funcţii cu număr variabil de parametri. Exemplul cel mai comun de astfel de funcţii sunt funcţiile din familia **printf()**, **scanf()**.

Definirea funcţiilor cu număr variabil de parametri

Prototipul funcţiilor cu număr variabil de parametrii arată în felul următor:

```
tip_rezultat nume_funcţie(listă_parametrii_fixaţi, ...);
```

Notaţia **„ ,... ”** comunică compilatorului faptul că funcţia poate primi un număr arbitrar de parametrii începând cu poziţia în care apare. De exemplu, prototipul funcţiei **fprintf()** arată în felul următor:

```
void fprintf(FILE*, const char*, ...);
```

Astfel, funcţia **fprintf()** trebuie să primească un pointer la o structură **FILE** şi un string de format şi eventual mai poate primi 0 sau mai multe argumente. Modul în care ele vor fi interpretate fiind determinat în cazul de faţă de conţinutul variabilei de tip **const char*** (acesta este motivul pentru care pot apărea erori la rulare în condiţiile în care încercăm să tipărim un număr întreg cu **%s**).

O funcţie cu număr variabil de parametrii trebuie să aibă cel puţin un parametru fixat, cu nume (motivul acestei limitări rezidă în modalitatea de implementare a funcţiilor cu număr variabil de parametri, după cum se va vedea în paragraful următor).

Implementarea funcţiilor cu număr variabil de parametri

Pentru prelucrarea listei de parametrii variabili este necesară includerea fişierului antet **stdarg.h**. Acesta conţine declaraţii pentru tipul de date variable argument list (**va_list**) şi o serie de macrodefiniţii pentru manipularea unei astfel de liste. În continuare este detaliat modul de lucru cu liste variabile de parametri.

- Declararea unei variabile de tip **va_list** (denumită de obicei **args**, **arguments**, **params**)
- Iniţializarea variabilei de tip **va_list** cu lista de parametri cu care a fost apelată funcţia se realizează cu macrodefiniţia **va_start(arg_list, last_argument)** unde:
  - **arg_list** reprezintă variabila de tip **va_list** prin intermediul căreia vom accesa lista de parametri a funcţiei.
  - **last_argument** reprezintă numele ultimei variabile fixate din lista de parametri a funcţiei (în exemplul următor, aceasta este şi prima variabilă, numită **first**).
- Accesarea variabilelor din lista de parametri se realizează cu macro-definiţia **va_arg(arg_list, type)**, unde
  - **arg_list** are aceeaşi semnificaţie ca mai sus.
  - **type** este un nume de tip şi reprezintă tipul variabilei care va fi citită. La fiecare apel se avansează în listă. În cazul în care se doreşte întoarcerea în listă, aceasta trebuie reiniţializată, iar elementul dorit este accesat prin apeluri succesive de **va_arg()**
- Eliberarea memoriei folosite de lista de parametri se realizează prin intermediul macro-definiţiei **va_end(arg_list)**, unde **arg_list** are aceeaşi semnificaţie ca mai sus.

Fie o functie care afiseaza un număr variabil de numere naturale primite ca parametru. Lista este terminată cu un număr negativ.

```
#include <stdio.h> // printf
#include <stdarg.h> // va_list, va_arg,
                    // va_start,va_end

// semnatuna functie
//  - functia nu intoarce nimic(void)
//  - are un numar variabil de parametri (...)
//  - primul parametru este fixat (int first)
void list_ints(int first, ...);

int main() {
    list_ints(-1); // nimic de afisat
    list_ints(128, 512, 768, 4096, -1); // afiseaza  pana la -1
    list_ints('a', 'b', 0xa, 0xb, -2); // afiseaza pana la -2; apel corect deoarece castul la int este valid
    list_ints(1, -1); // afiseaza pana la -1
    list_ints(1, 2, 3, -1, 1, 2, 3); // afiseaza pana la -1
    return 0;
}

void list_ints(int first, ...) {
    // tratam cazul special cand nu avem
    // nici un numar (in afara de delimitator)
    if (first < 0) {
    printf("No numbers present (besides terminator)\n");
    return;
    }
```

```
    // lista de parametri
    va_list args;

    /* initializam lista de parametri */
    va_start(args, first);

    // parcurgem lista de parametri pana ce
    // intalnim un numar negativ
    printf("These are the numbers (excluding terminator): ");
    int current = first;
    do {
        // afiseaza numarul curent
        printf("%d ", current);

        // obtine urmatorul numar intreg din lista
        current = va_arg(args, int);
    } while (current >= 0);
    printf("\n");

    /* curatam lista de parametrii */
    va_end(args);
}
```

Observati comportamentul acestei functii pentru diversi parametri.

```
gcc -Wall -Wextra test.c  -o test
./test

No numbers present (besides terminator)
These are the numbers (excluding terminator): 128 512 768 4096
These are the numbers (excluding terminator): 97 98 10 11
These are the numbers (excluding terminator): 1
These are the numbers (excluding terminator): 1 2 3
```

In exemplul anterior am considerat toate argumentele primite erau intregi. In general, acestea pot avea tipuri diferite.

Realizați o **funcție cu număr variabil de parametri**, numită de exemplu **show()** care să permită afisarea cu format a unui număr de variabile date ca parametri, acestia putand avea **tipuri diferite**.

Prototipul va arăta în modul următor:

```
int show(char *format, …);
```

- Pentru fiecare specificator de format, se va afisa variabila în variabila curentă un număr de octeți corespunzător tipului dat de specificator (vezi mai jos).
- Specificatorii de format suportați sunt următorii:
    - %c - dată de tip char
    - %d - dată de tip integer
    - %f - dată de tip float
    - %s - dată de tip char * (se va afisa un string)
- Pentru simplitate se va considera că după '%' urmează mereu unul din cei 3 specificatori.
- Functia va returna numarul de argumente procesate din lista.
- Afișați la stdout datele citite. In implementare aveti voie sa folositi functiile printf/fprintf.

```
#include <stdio.h> // printf
#include <string.h> // strlen
#include <stdarg.h> // va_list, va_arg,
                    // va_start,va_end

int show(const char *format, ...);

int main() {
    char name[] = "Gigel";
```

```
    int age = 15;
    float weight = 91.5;
    int argc;

    // afisare @ Gigel
    argc = show("%s are %d ani si %f de kg.\n", name, age, weight);

    // afisare numar de argumente folosite
    show("arguments used: %d/3\n", argc);

    // afisare double
    double x = 5.0;
    show("%f wtf?\n", x);

    return 0;
}


int show(const char *format, ...) {
    // n = lungime format, i iterator
    // proccesed = argumente procesate
    int n, i, proccesed;

    // initializari
    n = strlen(format);
    proccesed = 0;
    va_list args;
    va_start(args, format);

    // parcurge fiecare caracter din format
    for (i = 0; i < n; ++i) {
        // c e caracterul curent
        char c = format[i];

        // verifica sa nu inceapa o interpolare
        // de stringuri
        if (c != '%') {
            // afisez exact ce primesc
            printf("%c", c);
            continue;
        }

        // c == '%', deci am gasit o interpolare
        // decid ce caz am gasit
        ++proccesed;
        c = format[ ++i ];
        switch(c) {
            // am gasit "%d"
            // urmatorul parametru este un int
            case 'd': {
                int d = va_arg(args, int);
                printf("%d", d);
                break;
            }

            // am gasit "%f"
            // urmatorul parametru este un double
            case 'f': {
                double f = va_arg(args, double);
                printf("%f", f);
                break;
            }

            // am gasit "%s"
            // urmatorul parametru este un char*
            case 's': {
                char *s = va_arg(args, char*);
                printf("%s", s);
                break;
            }

            default:
                fprintf(stderr, "Asta nu trebuia sa se intample!\n");
                break;
        }
```

```
    }

    va_end(args);
    return proccesed;
}
```

Rulare:

```
gcc -Wall -Wextra test.c  -o test
./test

Gigel are 15 ani si 91.500000 de kg.
arguments used: 3/3
5.000000 wtf?
```

Incercati acasa sa inlocuiti functiile printf/fprintf cu alte primitive (de exemplu puts/putc).

- Codul a compilat, desi nu am implementat separat float si double. La pasarea parametrilor, unele tipuri de date sunt promovotate: float la double, char/short la int (stackoverflow [http://stackoverflow.com/questions/11270588/variadic-function-va-arg-doesnt-work-with-float]).

Exerciții de laborator CB/CD

Vă invităm să evaluați activitatea echipei de **programare CB/CD** și să precizați punctele tari și punctele slabe și sugestiile voastre de îmbunătățire a materiei. Feedback-ul vostru este foarte important pentru noi să creștem calitatea materiei în anii următori și să îmbunătățim materiile pe care le veți face în continuare.

Găsiți formularul de feedback în partea dreaptă a paginii principale de programare de pe cs.curs.pub.ro într-un frame numit "FEEDBACK" (moodle [http://cs.curs.pub.ro/2016/course/view.php?id=17]). Trebuie să fiți inrolați la cursul de programare, altfel veți primi o eroare de acces.

Pentru a putea înțelege și valorifica feedback-ul mai ușor, apreciem orice formă de feedback negativ constructivă. Nu este suficient să ne spuneți, spre exemplu, *tema 5 a fost grea*, ne dorim să știm care au fost dificultățiile și, eventual, o propunere despre cum considerați că ar fi trebuit procedat.

Primul exercitiu presupune modificarea/adaugarea de instructiuni unui cod existent pentru a realiza anumite lucruri. In momentul actual programul populeaza vectorul cu valorile parametrilor primiti de functie (atat timp cat valoarea parametrului este > 0). Daca nu s-a populat tot vectorul, se adauga 0 pe pozitiile ramase goale.

ex1.c

```
#include <stdio.h>
#include <stdarg.h>

#define N 10

void insert_elements(int *v, int n, ...)
{
    int count = 0;

    va_list args;
    va_start(args, n);

    int elem = va_arg(args, int);
    while (elem > 0 && count < n) {
        v[count++] = elem;
        elem = va_arg(args, int);
    }

    while (count < n) {
        v[count++] = 0;
    }
}
```

```
int main(void)
{

    int v[N];
    int i;

    insert_elements(v, N, 1, 2, 3, 4, -1);

    for (i = 0; i < N; i ++) {
      printf("%d ", v[i]);
    }

    printf("\n");

    return 0;
}
```

Cerinte:

- Sa se modifice functia astfel incat sa se poata popula si un vector cu elemente de tip char. Functia va fi declarata astfel:
  - **void insert_elemenets(void *v, int n, char type, ...)** - unde type va specifica tipul (‘d’ pentru int si ‘c’ pentru char).
  - Daca nu se umple tot vectorul cu valori, vor trebui puse niste valori default (cum este si in codul deja existent): pentru int va ramane valoarea 0, iar pentru char va fi ‘a’.
- In functie de simbolul NUMBERS, programul va popula fie un vector de numere, fie un vector de caractere si il va afisa (se pot lasa valori specifice trimise la functia de populare)
  - Simbolul nu trebuie definit in cadrul fisierului sursa

Următoarele două probleme vă vor fi date de asistent în cadrul laboratorului.

Exerciţii de Laborator

Exercițiile următoare fac referire la anumite exemple (ex. **Exemplu1**). Aceste exemple se găsesc în laborator (CTRL + F pentru **Exemplu1**).

1. [1p] Completati formularul de **FEEDBACK** pentru Programarea calculatoarelor (CA) [https://acs.curs.pub.ro/2018/course/view.php?id=127]. Vă rugăm să îl completați cu atenție și răbdare. Apreciem orice feedback **NEGATIV CONSTRUCTIV** . Vă rugăm nu spuneți **DOAR** lucruri precum “tema X a fost grea”, “nu am reușit să fac tot la parțial”, “Gigel apare de prea multe ori prin laboratoare”. Încercați să argumentați de ce credeți că s-a ajuns în acea situație și (eventual) cum credeți voi că am putea remedia asta pentru anul viitor. Mulțumim!
2. [1p] Să se realizeze un program care să afișeze toți parametrii liniei de comandă primiți. Hint: **Exemplu1**.
3. [2p] Scrieti un program C care primeste argumente in linia de comanda. Acesta trebuie sa respecte urmatoarele reguli.
   - Numele executabilului va fi `gigel`.
   - Primeste cel putin `doua argumente` in linia de comanda. In caz contrar, se va afisa la stderr mesajul `Gigele, cel putin doua!`, iar programul se va termina cu codul de iesire `-2`.
   - Lista de argumente reprezinta un sir de numere naturale (`unsigned int`). Se cere sortarea listei folosind functia `qsort` [http://www.cplusplus.com/reference/cstdlib/qsort/?kw=qsort].
   - Daca lista de argumente contine stringuri care nu reprezinta numere intregi fara semn, se va afisa la stderr mesajul `Gigele, da-mi numere naturale!`. In acest caz, programul va

iesi cu valoarea INT_MIN [http://www.cplusplus.com/reference/climits/].

4. [2p] Realizați un macro SWAP(x, y). Hint: Puteti folosi sizeof pentru a determina cata memorie este necesara pentru variabila de interschimbare. De asemenea, puteti consulta si **Exemplu 5**.
5. [4p] Realizați implementarea funcției cu număr variabil de parametri numită gcd, care să permită aflarea celui mai mare divizor comun (gcd [https://en.wikipedia.org/wiki/Greatest_common_divisor]) al parametrilor dați (cel puțin două elemente care sunt numere naturale). Puteti presupune ca lista se termina cu un numar negativ. Hint: **Exemplu8**.

Bonus

[4p] mycat (PC CA - 2015). Implementaţi comanda mycat [+-][n][LCB] nume_fisier, având ca efect scrierea a n unităţi din fişierul text specificat la unitatea standard de ieşire. O unitate poate reprezenta: o linie (L), un caracter (C) sau un bloc (B = 512 caractere).

- Dacă primul argument este + se vor afişa primele n unităţi din fişier, iar dacă este – atunci se afişează ultimele n unităţi. Cu excepţia numelui fişierului, oricare din elementele comenzii poate lipsi, caz in care se consideră valorile implicite: + pentru primul argument, 10 pentru argumentul 2 şi L pentru argumentul 3.
- Exemple:

```
> ''mycat date.txt'' – afişează primele 10 linii din fişier
```

```
> ''mycat - date.txt'' – afişează ultimele 10 linii din fişier
```

```
> ''mycat -25 C date.txt'' – afişează ultimele 25 de caractere din fişier
```

```
> ''mycat +3 B date.txt'' – afişează primele 3 blocuri din fişier
```

- Indicaţie: În cazul în care se cere afişarea ultimelor n linii din fişier, se va determina numărul de linii din fişier nl şi se vor ignora primele nl-n linii. Dacă se cere afişarea ultimelor n caractere, va trebui să contorizăm numărul de caractere din fişier.
- NU se vor folosi variabile globale.

- Punctaje:
  - Pentru implementarea corecta si completa a tuturor functionalitatilor corespunzatoare unui argument se acorda cate 1p.
  - Se acorda inca 0.5p daca rularea respecta formatul de mai sus. Hint: Linux alias command [http://alvinalexander.com/blog/post/linux-unix/create-aliases].
  - Se acorda inca 0.5p daca implementarea propusa este performanta. Justificati alegerea facuta!

Probleme laborator 14:00 - 16:00 [https://docs.google.com/document/d/1lwQBmJ4YuaBmpUOb5uUIs27TwLvWu-6jTL3aQ9TAwIA/edit?usp=sharing]