

Complemente de programare 1

C01: Pointeri la funcții. Make & Makefile

conf. dr. ing. Elena Șerban șef lucr. dr. ing. Alexandru Archip

Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare

Calculatoare și Tehnologia informației (licență, an I) – 2022 – 2023

E-mail: elena.serban@academic.tuiasi.ro | alexandru.archip@academic.tuiasi.ro

Noțiuni avansate de programare în C

Continuare a disciplinei Programarea Calculatoarelor

Unelte

- Linux (recomandat Centos) sau macOS (pe calculatoarele din laborator)
- vim (editare de texte), fișier de configurare - .vimrc (exemplu în *Tutorial Vim* de la PC)
- gcc (pentru compilări, linkeditări)
- gdb, valgrind, ...

Nota finală

- 50% - test final pe moodle
- 50% - nota de la laborator

- 1 Pointer la un tablou
- 2 Pointer la o funcție
- 3 Interpretarea declarațiilor complexe în C
- 4 Utilitarul *make*. Fișiere *Makefile*

Cuprins

- 1 Pointer la un tablou
- 2 Pointer la o funcție
- 3 Interpretarea declarațiilor complexe în C
- 4 Utilitarul *make*. Fișiere *Makefile*

Definirea unui pointer la un tablou

Folosind sinonime

```
typedef int iVECTOR10[10];  
.....  
iVECTOR10 a;  
  
iVECTOR10 v[5];  
  
iVECTOR10 *pv = 0;  
  
pv = &v[0];      //pv = v;
```

Definirea directă

Dacă nu folosim typedef pentru definirea unui pointer la un tablou se va scrie:

```
int (*pv)[10] = 0;
```

Alocarea dinamică pentru un pointer la un tablou

Se consideră M o constantă simbolică definită cu ajutorul directivei `define`. Definirea unui pointer `pv` la un tablou cu M elemente de tipul T este:

```
T (*pv)[M] = 0;
```

Pentru acest pointer trebuie să facem alocare dinamică de memorie:

```
pv = (T (*)(M)) malloc(N * sizeof(*pv));  
if(0 == pv)  
{  
    fprintf(stderr, "Alocare de memorie esuata\n");  
    exit(EXIT_FAILURE);  
}
```

Alocare dinamică pentru un pointer la T

```
T *p = 0;  
//Urmeaza alocarea dinamica pentru pointer si obtinem  
// situatia din figura.
```

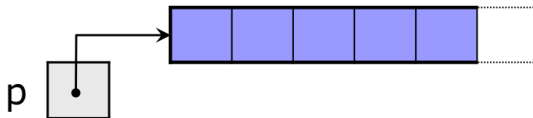


Figura 1. Alocare dinamică pentru un pointer la T

Alocare dinamică pentru un pointer la un tablou

```

T (*pv)[M] = 0;
.....
pv = (T (*)[M]) malloc(1 * sizeof(*pv));
if(0 == pv)
{
    fprintf(stderr, "Alocare de memorie esuata\n");
    exit(EXIT_FAILURE);
}

```

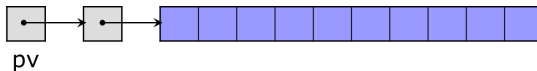


Figura 2. Alocare dinamică pentru un pointer la un tablou de **M** elemente de tip **T**

Alocare dinamică pentru un pointer la un tablou

```

T (*pv)[M] = 0;
.....
pv = (T (*)[M]) malloc(N * sizeof(*pv));
if(0 == pv)
{
    fprintf(stderr, "Alocare de memorie esuata\n");
    exit(EXIT_FAILURE);
}

```

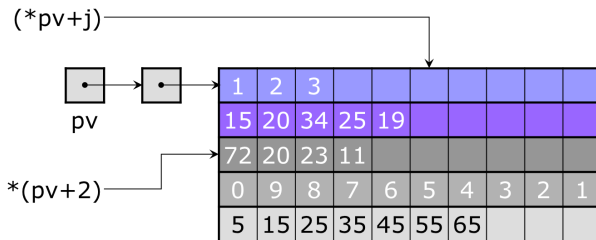


Figura 3. Alocare dinamică pentru un tablou de tablouri

Pointer la un tablou

Pentru stocarea vectorilor bidimensionali putem folosi:

- (1) matrice statică
- (2) pointer la pointer
- (3) tablou de pointeri
- (4) pointer la un tablou

Sunt echivalente (1) cu (4) și (2) cu (3). Pentru toate aceste tipuri de date, accesul la un element se poate face prin folosirea a doi indcși (toate modelează de fapt tablouri bidimensionale).

Vectori cu dimensiune variabilă

Variable length arrays - VLA - există în **C**, nu există în **C++**. Următorul exemplu este preluat din [2]:

```
#include <stddef.h>           //pentru size_t

size_t fsize3(int n)
{
    char b[n+3];              // variable length array
    return sizeof b;          // execution time sizeof
}

int main(void)
{
    size_t size;
    size = fsize3(10);        // fsize3 returns 13
    return 0;
}
```

Listing 1. Exemplu 1 de folosire a unui VLA

Vectori cu dimensiune variabilă

Tot din [2] este și următorul exemplu de folosire a unui vector cu dimensiune variabilă:

```
{
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a;    // p == &a[0]
    p += 1;             // p == &a[1]
    (*p)[2] = 99;        // a[1][2] == 99
    n = p - a;           // n == 1
}
```

Listing 2. Exemplu 2 de folosire a unui VLA

Cuprins

- 1 Pointer la un tablou
- 2 Pointer la o funcție
- 3 Interpretarea declarațiilor complexe în C
- 4 Utilitarul *make*. Fișiere *Makefile*

Lansarea în execuție a unui program

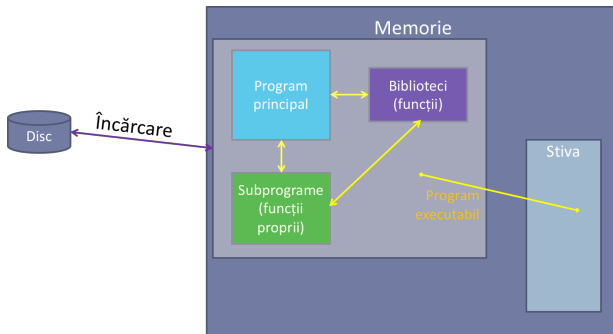


Figura 4. Lansarea în execuție a unui program

Fiecărei funcții din program i se asociază o adresă de memorie (inclusiv funcției **main** și funcțiilor de bibliotecă).

Putem extrage această adresă într-o variabilă de tip pointer.

Definirea unei variabile de tip pointer la funcție

Definirea unei variabile de tip pointer la funcție

```
tip (*pf)(tip1 p1, tip2 p2, ..., tipn pn);
```

sau

```
tip (*pf)(tip1, tip2, ..., tipn);
```

Observație

Și la declararea unei funcții putem scrie doar tipul parametrilor fără nume:

```
int citesteVector (int *, int);
```

Definirea unei variabile de tip pointer la funcție

Exemple

```
int (*pf)(float a, int b);  
int* (*pf2)(int n);
```

De comparat cu

```
int *pf(float a, int b);  
int* *pf2(int n);
```

Observație: un pointer la pointer este echivalent cu un tablou de pointeri.

Definirea unui sinonim la un pointer la o funcție

Dacă se definește un sinonim la pointerul la funcție pe care îl vom folosi:

```
typedef int (*FPTR)(float a, int b);
```

atunci definiția

```
FPTR pf;
```

este echivalentă cu:

```
int (*pf)(float a, int b);
```

Exemplu de folosire a unui pointer la o funcție

```
int f1(float a, int b);  
int f2(float a, int b);  
int f3(float a, int b);  
  
pf = &f1;
```

Listing 3. Exemplu 1

```
if(test)  
{  
    pf = &f2;  
}  
else  
{  
    pf = &f3;  
}
```

Listing 4. Exemplu 2

Exemplu de folosire a unui pointer la o funcție

```
pf = test ? &f2 : &f3  
sau  
pf = test ? f2 : f3
```

Listing 5. Exemplu 3

```
x = (*pf)(3.14, 5);    //sau x = pf(3.14, 5);
```

Listing 6. Apelul unei funcții folosind un pointer la funcție

Pointer la funcție ca argument al unei funcții

Fie o funcție g :

```
tip_g g(lista_de_parametri_g);
```

și o funcție f care are ca parametri un întreg și un pointer la o funcție de tip g .
Prototipul funcției f este:

```
tip_f f(int a, tip_g (*pg)(lista_de_parametri_g));
```

sau echivalent:

```
tip_f f(int, tip_g (*)(lista_de_parametri_g));
```

Calcularea integralei unei funcții prin metoda trapezelor

Fie o funcție

$$f : [a, b] \rightarrow \mathbb{R}$$

Folosind limbajul **C**, să se scrie o funcție care calculează:

$$\int_a^b f(x) dx$$

Semnificația integralei: integrala unei funcții este aria cuprinsă între graficul funcției, dreptele $x = a$, $x = b$ și axa Ox .

Fie o diviziune echidistantă a intervalului $[a, b]$:

$$a = x_0 < x_1 < \dots < x_n = b \quad (1)$$

Această diviziune împarte intervalul $[a, b]$ în n subintervale $[x_r, x_{r+1}]$, fiecare dintre aceste subintervale având o lungime dx dată de relația:

$$dx = \frac{b - a}{n} \quad (2)$$

Calcularea integralei unei funcții prin metoda trapezelor

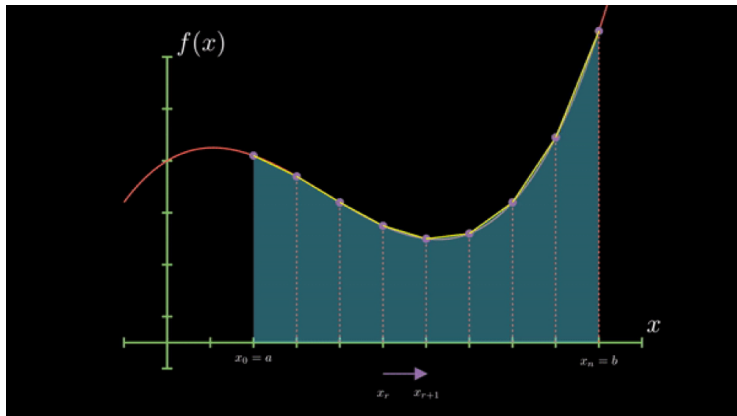


Figura 5. Calculul integralei prin metoda trapezelor [3]

Calcularea integralei unei funcții prin metoda trapezelor

Calculul ariei unui trapez (Figura 5) se face cu formula :

$$A_r = \frac{f(x_r) + f(x_{r+1})}{2} \cdot dx, \quad r = \overline{0, n-1} \quad (3)$$

Aria totală (integrala definită) este calculată cu formula:

$$\int_a^b f(x) dx = \sum_{r=0}^{n-1} A_r = \left(\frac{f(a) + f(b)}{2} + \sum_{r=1}^{n-1} f(x_r) \right) \cdot dx \quad (4)$$

Pentru implementare avem nevoie de două funcții:

- funcția care calculează valoarea unei funcții într-un punct $f(x_r)$;
- funcția care calculează valoarea integralei definite pe intervalul $[a,b]$ folosind formula (4).

Pointer la funcție ca argument al unei funcții

Prototipurile celor două funcții sunt:

```
double f(double);  
  
double integralaTrapez(  
    double a,  
        //Limita inferioara a intervalului de integrare  
    double b,  
        //Limita superioara a intervalului de integrare  
    unsigned int n,  
        //Numarul de subintervale din diviziune  
    double (*pf)(double)  
        //Functia de integrat  
);
```


Pointer la funcție ca argument al unei funcții

Funcția care calculează integrala este:

```
double integralaTrapez(double a, double b,
                      unsigned int n,
                      double (*pf)(double))
{
    double sum; //Valoarea calculata a integralei
    double x;   //Un punct din diviziune
    double dx;  //Lungimea unui subinterval
    double eps = 1e-4;

    dx = (b - a)/n;

    sum = ((*pf)(a)+(*pf)(b)) / 2;

    for(x = a + dx; fabs(x + dx - b) > eps; x += dx)
    {
        sum += ((*pf)(x));
    }

    sum *= dx;

    return sum;
}
```

Pointer la funcție ca argument al unei funcții

Exemplu:

Pentru calcularea valorii $v1$ a integralei:

$$\int_{-1}^1 \sin(x) dx$$

Apelul este:

```
v1 = integralaTrapez(-1, 1, 100, sin);
```

iar pentru valoarea $v2$ a integralei

$$\int_2^3 \sin(e^{2x} + 3) dx$$

apelul este:

```
v2 = integralaTrapez(2, 3, 100, f);
```

unde f este funcția care calculează valoarea funcției de integrat într-un punct al diviziunii considerate.

Tablouri de pointeri la funcții

Pointerii la funcții se pot grupa în tablouri.

Exemplu

Un tablou de 10 pointeri la funcții care primesc ca parametru un real în dublă precizie și returnează un real în dublă precizie este definit astfel:

```
double (*fm[10])(double);
```

Apelul unei funcții din acest tablou se face astfel:

```
x = (*fm[3])(2.45);
```

Problemă

Pentru că în loc de un tablou unidimensional putem avea un pointer la tipul respectiv de date, cum definim un pointer la un pointer la o funcție reală de argument real? Cum facem alocare dinamică pentru acest pointer?

Pointeri la funcții ca membri în structuri

Pointerii la funcții pot apărea și ca membri în structuri. Astfel de construcții pot fi utile pentru:

- Construirea unor meniuri
- Aplicarea unor principii de programare orientată pe obiecte în aplicații dezvoltate folosind limbajul **C** standard.

Cuprins

- 1 Pointer la un tablou
- 2 Pointer la o funcție
- 3 Interpretarea declarațiilor complexe în C
- 4 Utilitarul *make*. Fișiere *Makefile*

Regula dreapta-stânga sau Regula spiralei în sensul acelor de ceas

Pentru citirea și construirea definițiilor de variabile, în limbajul **C** se folosește așa numita regulă dreapta-stânga (sau regula spiralei în sensul acelor de ceas) [6][7]. La citirea unei definiții se pornește de la numele variabilei și apoi se interpretează simbolurile din dreapta, apoi se trece în stânga, din nou în dreapta ș.a.m.d.

Exemple:

```
int (*tab)[10];

int (*f)[*];
/*
 * un pointer la un VLA cu un numar nespecificat
 * de intregi
 */
```

Regula dreapta-stânga sau Regula spiralei în sensul acelor de ceas

Exemple:

```
int *(*f[10])(int, int (*)(int, int []));
```

Pornim de la numele variabilei

La dreapta [10]

La stânga (*)

La dreapta)(int,

int (*)

(int, int[]))

La stânga int *

f

este un tablou de 10 elemente de tip

pointeri la

funcții care primesc ca parametri un întreg și

un pointer la o funcție care returnează un întreg și

primește ca parametri un întreg și un tablou de întregi

și f returnează un pointer la int

Regula dreapta-stânga sau Regula spiralei în sensul acelor de ceas

```

int (*)(*);
/*
 *  tipul de data pointer la VLA cu un numar
 *  nespecificat de intregi
 */
int (*const [])(unsigned int , ...);
/*
 *  un tablou cu un numar nespecificat de pointeri
 *  constanti la functii variadice in care primul
 *  parametru este un intreg si care returneaza un intreg
 */

int * (* (*fp1) (int) ) [10];

int * ( * ( *arr[5]) (void) ) (void);

void (* signal( int , void (*fp)(int) ))(int);

```

```

void (* signal( int, void (*fp)(int) ))(int);

```


Regula dreapta-stânga sau Regula spiralei în sensul acelor de ceas

```
int * (* (*fp1) (int) ) [10];
```

Pornim de la numele variabilei: **fp1** este un pointer la o funcție care are ca parametru un întreg și returnează un pointer la un tablou de 10 elemente, fiecare element fiind un pointer la întreg.

```
int * ( * ( *arr[5]) (void) ) (void);
```

Pornim de la numele variabilei: **arr** este un **tablou de 5 elemente**, fiecare element este **un pointer la o funcție** care nu are parametri și **returnează un pointer la o funcție** care nu are parametri și returnează un pointer la întreg.

```
void (* signal(int, void (*fp)(int)) )(int);
```

Pornim de la identificatorul principal: **signal** este o funcție care primește ca parametri un întreg și un pointer la funcție (**fp**) care primește ca parametru un întreg și nu returnează nimic. Funcția **signal** returnează un pointer la o funcție care primește ca parametru un întreg și nu returnează nimic.

Cuprins

- 1 Pointer la un tablou
- 2 Pointer la o funcție
- 3 Interpretarea declarațiilor complexe în C
- 4 Utilitarul *make*. Fișiere *Makefile*

Utilitarul *make*

Gestionarea fișierelor care compun un proiect.

Mentenanța programelor complexe: *Makefile* - mecanism Unix

Un *makefile* este un fișier (script) care conține:

- Structura proiectului (fișiere, dependențe)
- Instrucțiuni pentru crearea fișierelor

La lansarea în execuție *make* citește fișierul *Makefile*, înțelege structura proiectului și creează executabilul.

Acest mecanism nu este limitat la programe scrise în **C**.

Structura proiectului și dependențele pot fi reprezentate ca un *DAG* (Directed Acyclic Graph)

Utilitarul *make*

- Exemplu: un program care conține trei fișiere:
 - Fișierul cu funcția main - p0.c
 - Fișierul cu funcții - operatii.c
 - Fișierul header propriu - operatii.h
- operatii.h este inclus în ambele fișiere cod-sursă
- executabilul se va numi p0

Utilitarul *make*

```
p0: p0.o operatii.o
    gcc -Wall p0.o operatii.o -o p0

p0.o: p0.c operatii.h
    gcc -c -Wall p0.c

operatii.o: operatii.c operatii.h
    gcc -c -Wall operatii.c
```

Utilitarul *make*

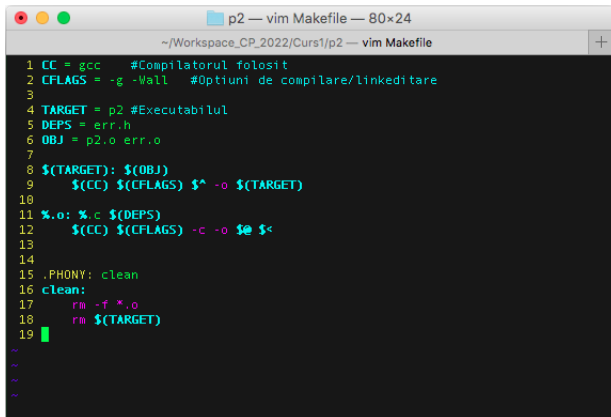
Un fișier *Makefile* cuprinde o serie de reguli. O regulă este formulată astfel:

```
targets : prerequisites  
    command  
    command  
    . . . . .
```

targets: nume de fișiere separate prin spații sau numele unei acțiuni care trebuie realizată (*phony targets*)

prerequisites: nume de fișiere separate prin spații. Fișierele trebuie să existe înainte de a fi folosite. Se mai numesc și *dependencies* - dependențe.

command: comenzile necesare pentru realizarea țintelor (*targets*)

Utilitarul *make*

```
1 CC = gcc      #Compilatorul folosit
2 CFLAGS = -g -Wall  #Optiuni de compilare/linkeditare
3
4 TARGET = p2 #Executabilul
5 DEPS = err.h
6 OBJ = p2.o err.o
7
8 $(TARGET): $(OBJ)
9     $(CC) $(CFLAGS) $^ -o $(TARGET)
10
11 %.o: %.c $(DEPS)
12     $(CC) $(CFLAGS) -c -o $@ $<
13
14
15 .PHONY: clean
16 clean:
17     rm -f *.o
18     rm $(TARGET)
19
~
~
~
```

Figura 7. Fișier makefile 3

Utilitarul *make*

variabile automate

`$@` - ținta (ceea ce se găsește pe prima linie a unei reguli înainte de :)

`$<` - primul termen din lista de dependențe

`^` - dependențele, separate prin spații, fără duplicate

`$(word 2,$^)` - al doilea termen din lista de dependențe

Resurse utile

- ❶ *******, ISO/IEC 9899
- ❷ *******, ISO/IEC 9899:2018 (aka C17 and C18)
- ❸ Kazi Abu Rousan, Trapezoidal Rule: A Method of Numerical Integration, Jul, 2020
- ❹ *******, Learn Makefiles
- ❺ *******, GNU make
- ❻ Anderson, David, The Clockwise/Spiral Rule, 1994
- ❼ Punathambekar, V. A., How to interpret complex C/C++ declarations, Jul, 2004