Programarea Calculatoarelor
Cursul 4

Funcții
Tablouri

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

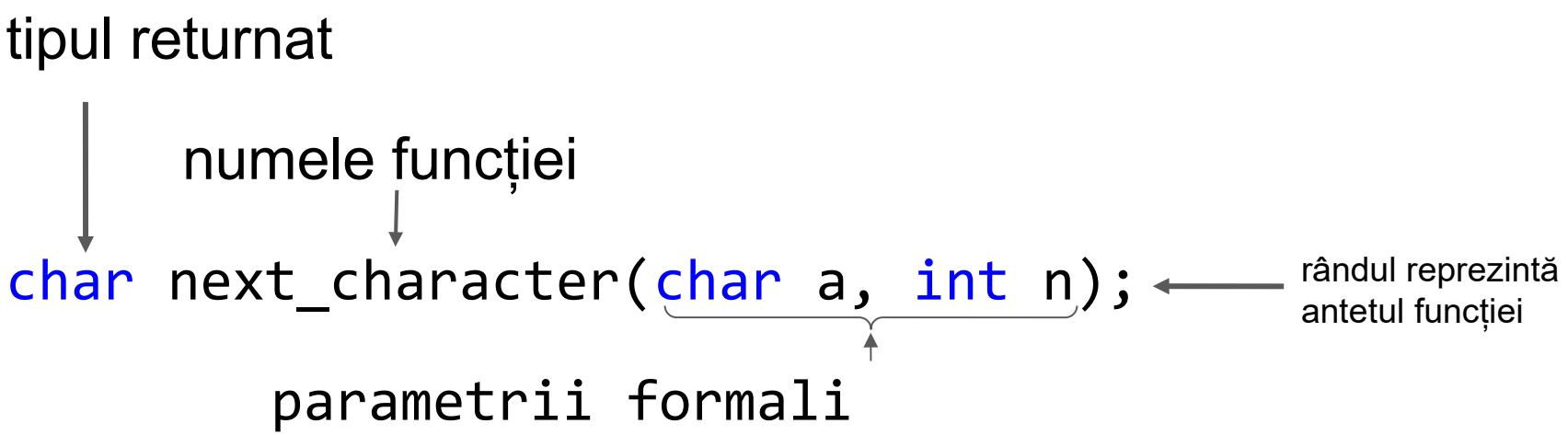
- Cum scriem funcții?
  - Când este recomandat să le folosim?
  - Ce este diferit față de cod scris în main?
- Cum lucrăm cu un șir de variabile?

Funcții - introducere

- O funcție este unitate de sine stătătoare care primește date de intrare prin parametri, realizează calcule și returnează rezultatul
- Este un mini-program
- Funcțiile care nu au efecte secundare (nu modifică variabile externe funcției) și produc același rezultat pentru aceiași parametrii sunt funcții pure = programare funcțională
- Permit modularizarea programului
  - sunt reutilizabile
  - produc cod scurt și ușor de citit

Declarația funcțiilor

- O funcție se declară prin antetul funcției sau header (eng.)
- Antetul poate fi urmat de ;
  - în acest caz avem doar declarația care arată cum comunică funcția cu restul programului dar nu și ce operații face = prototip
- Antetul poate fi urmat de corpul funcției încadrat de { }
  - în acest caz avem atât declarația cât și definiția funcției



Tipul returnat

- Trebuie specificat (obligatoriu)
  - dacă nu e prezent este asumat ca int
- Poate fi orice tip din limbaj
  - tablouri nu - mai multe la pointeri
- Poate fi lipsa unui tip specificat prin void
  - funcția nu returnează nimic = este o procedură
  - o folosim pentru citire, afișări sau alte efecte secundare

```
void interschimba(int a, int b);
```

Numele, parametrii formali, apelul funcției

- Numele poate fi orice identificator valid
  - fără spațiu - putem folosi _
  - să fie sugestiv, dar nu foarte lung
- Parametrii formali se definesc la fel ca variabilele
  - pentru fiecare trebuie să apară tipul și numele
  - `f(int a, b)` este incorect - `f(int a, int b)` este corect
  - în cazul unui prototip, pot să lipsească denumirile `f(int, int);`
- Apelul funcției se realizează prin numele funcției, paranteze rotunde și expresii pentru fiecare parametru = argumente
  - instrucțiunea `f(a, b)` apelează funcția `f` cu argumentele `a` și `b`
  - funcția trebuie să fie declarată în cod înainte de a o apela
  - definiția poate să apară și după apel

Definiția funcției

- Se poate realiza pe loc, după declarație
- Se poate realiza separat
- Reprezintă instrucțiunile care formează corpul funcției
- Se încadrează între acolade { }
- Pentru funcții care returnează (non-void)
  - trebuie să existe instrucțiunea `return` pe fiecare ramură de execuție
  - instrucțiunea `return` iese din funcție și returnează valoarea expresiei care urmează
  - tipul expresiei este implicit convertit la tipul returnat
- Pentru funcții void
  - instrucțiunea `return` fără expresie poate fi prezentă (opțional)
  - instrucțiunea `return` iese din funcție

Funcții - exemplu simplu

```c
#include <stdio.h>
int ultim_bit(int x){
    if (x&1)
        return 1;
    else
        return 0;
}

int main(){
    int b = ultim_bit(5);

    printf("%d\n", b);
    printf("%d\n", ultim_bit(2));
    return 0;
}
```

- se declară și se definește funcția
- tipul returnat este `int`, așteaptă un singur parametru `int`
- pe fiecare ramură de execuție avem o instrucțiune `return`
- cele două sunt echivalente cu un singur `return x&1`

- apelul funcției - funcția trebuie să fie declarată în prealabil
- preluăm rezultatul apelului și îl stocăm în variabila b
- putem afișa direct rezultatul apelului

Funcții - exemplu complex

```c
#include <stdio.h>
void binar_invers(int);

int main(){
   binar_invers(123);
   return 0;
}

void binar_invers(int x){
   while(x){
      printf("%d", x&1);
      x >>= 1;
   }
   return;
   puts("dupa return");
}
```

- se declară funcția - doar prototip, pot lipsi denumirile
- tipul returnat este `void`, așteaptă un singur parametru `int`

- se apelează funcția, nu returnează nimic
- pentru apel trebuie să cunoaștem doar prototipul

- definiția funcției - poate fi după apel
- cât timp x este diferit de 0
- se afișează ultimul bit (cel mai nesemnificativ)
- se șterge ultimul bit prin shiftare

- se termină funcția
- instrucțiunile după `return` nu se execută
- dacă nu apare `return`, atunci funcția se termină după ce se execută toate instrucțiunile din corpul ei

Trimitere prin valoare

- Argumentele trimise la funcții se trimit prin valoare
  - se copiază valorile
  - dacă argumentele sunt variabile ele nu se vor schimba din cauza apelului
  - tot ce se întâmplă în interiorul funcției nu afectează exteriorul
- Dacă vrem să modificăm argumentele trimise avem nevoie de alt mecanism

Trimitere prin valoare - exemplu

```c
#include <stdio.h>
void interschimba(int a, int b){
     int t = a;
     a = b;
     b = t;
     printf("in func a = %d, b = %d\n", a, b);
}

int main(){
     int a = 1, b = 2;
     printf("inainte a = %d, b = %d\n", a, b);
     interschimba(a, b);
     printf("dupa    a = %d, b = %d\n", a, b);
     return 0;
}
```

- se declară și se definește funcția
- folosim variabila temporară `t` care e valabilă doar în funcție pentru a interschimbă argumentele `a` și `b`
- a = 2 și b = 1

- a = 1 și b = 2
- se trimite a și b prin valoare
- a = 1 și b = 2
- am trimis la funcție doar o copie a valorilor stocate în a și b

Studiu - funcții din bibliotecile stdlib.h, math.h

```
int rand( );
```

- returnează un număr pseudo-aleator din intervalul

[0, RAND_MAX], RAND_MAX este de obicei 32767

```
double pow( double base, double exponent );
```

- funcția putere pentru numere flotante (double)
- Atenție! - nu dă rezultat corect pentru numere întregi

```
float  atan2f( float y, float x );
```

- funcția returnează arctangenta valorii y/x în cadranul corect pe baza perechii x, y
- unghiul returnat este în radiani în domeniul [-pi, pi]

Consultați https://en.cppreference.com/w/c/numeric/math

Funcția pow - greșeală comună

```c
#include <stdio.h>
#include <math.h>

int main(){
    long long x = pow(10, 18) - 100;
    printf("%lld\n", x);
    return 0;
}
```

- includem biblioteca math.h

- Funcția `pow` așteaptă argumente de tip `float` sau `double`
- Argumentele trimise, 10 și 18, sunt convertite implicit la `double`
- Rezultatul returnat de funcție este tot `double`
- `double` are precizie de aproximativ 15 cifre
- Ultimele 3 cifre sunt greșite
- Afișează 9999999999999872
  - depinde de setări de compilator

- Se recomandă evitarea funcției `pow` dacă se dorește rezultat exact pe numere întregi

Stiva de execuție - eng. execution stack

- Este o zonă de memorie pentru stocarea variabilelor locale, ale adreselor de revenire din funcții și ale argumentelor
- Este o stivă fiindcă la orice apel de funcție se introduc valori noi în vârful stivei și la revenire se extrage tot din vârful stivei
- Această zonă are dimensiuni reduse de câțiva MB
- Trebuie să ne asigurăm că nu depășim limita
  - să folosim variabile locale de dimensiune mică
  - să schimbăm dimensiunea stivei prin setări la linker

Apelul funcțiilor

- Etapele principale ale apelului unei funcții și revenirea din apel în funcția părinte
  - Argumentele apelului sunt evaluate și trimise funcției
  - Adresa de revenire este salvată pe stivă
  - Controlul trece la funcția care este apelată
  - Funcția apelată alocă pe stivă spațiu pentru variabilele locale și pentru cele temporare
  - Se execută instrucțiunile din corpul funcției
  - Dacă există valoare returnată, aceasta este salvată
  - Spațiul alocat pe stivă este eliberat
  - Utilizând adresa de revenire controlul este transferat în funcția care a inițiat apelul

Depășirea stivei - eng. stack overflow

```
int f(int n){
     if (n == 0)
          return 0;
     return n + f(n-1); //B
}

int main(){
     f(10000000); //A
     return 0;
}
```

- funcție recursivă = se autoapelează
- condiție de oprire

- rândul unde trebuie revenit după apel
- după revenire se calculează suma și se returnează rezultatul

f apel 4 - n = 0
f apel 3 - n = 1, apel recursiv, revenire la B
f apel 2 - n = 2, apel recursiv, revenire la B
f apel 1 - n = 3, apel recursiv, revenire la B
main, apel f, revenire la A

Stiva de execuție pentru *n = 3*

- f este apelat la rândul (adresa) *A* din main
- f se auto-apelează și se salvează valoarea locală a argumentului *n* și a adresei de revenire *B*
- când *n = 0* funcția returnează
- se eliberează variabilele din vârful stivei

Dacă *n* inițial este mare atunci se umplă stiva

Cel mai mare divizor comun - revizitat

- Vom dezvolta un algoritm mai eficient
- Utilizăm proprietatea:
  - Dacă a și b au un divizor comun d, atunci și diferența lor este divizibilă cu d
  - a = xd, b = yd atunci a-b = xd - yd = (x-y)d care este divizibil cu d
  - Dacă a = b atunci cmmdc este tot a
- Propunem următorul algoritm

```
cat timp a diferit de b
 daca a > b
    a = a-b
 altfel
    b = b-a
d = a
```

Cel mai mare divizor comun - revizitat

- Algoritmul anterior este mai eficient dacă numărul mai mic este mai mare ca 1 fiindcă facem pași mai mari ca 1
- Dacă a sau b este 0 algoritmul nu se termină
- Altfel se termină fiindcă
  - 1 este divizor comun pentru orice pereche
  - se descrește fie a fie b cu cel puțin 1 la fiecare pas
- Putem comprima mai mulți pași în unul singur
  - Dacă a > b, se scade b din a în mod repetat până când a devine mai mic ca b (exemplu a = 10, b = 4, scădem b de 2 ori)
  - Noua valoare va fi a' = a - qb, a' < b (a' = 10 - 2*4 = 2)
  - Atunci q trebuie să fie câtul împărțirii a/b și a' restul împărțirii
  - Rezultă că putem înlocui perechea (a, b) cu (b, a%b)
  - Dacă b este 0 atunci răspunsul este a
- Rezultă algoritmul lui Euclid
  - foarte eficient - timp $O(\log_{\varphi} a)$

Cel mai mare divizor comun - revizitat

- Algoritmul lui Euclid
  - funcționează și dacă a = 0 sau b = 0 sau ambele
  - dacă a < b atunci în primul pas se interschimbă a cu b

```
cat timp b diferit de 0
 r = a % b
 a = b
 b = r
d = a
```

Cel mai mare divizor comun - revizitat - implementare

```
int gcd(int a, int b){
    while(b != 0){
        int r = a%b;
        a = b;
        b = r;
    }
    return a;
}
```

```
cat timp b diferit de 0
    r = a % b
    a = b
    b = r

d = a
```

- Verificați corectitudinea
- Analizați și cazurile speciale a sau b 0
- Comparați timpul de execuție pentru numere mari ~ 10^9 cu algoritmii precendenți
- Care este cazul cel mai nefavorabil pentru algoritmul lui Euclid?

Variabile tablou - introducere

- O variabilă de tip tablou este format din mai multe elemente aflate unul după celălalt în memorie
- Foarte des avem nevoie de variabile care formează un șir
  - De exemplu, șirul x_i format din biții lui x
- Există o ordine bine definită a elementelor
- În loc să folosim câte o variabilă x1, …, xn putem să definim un tablou (șir) de variabile
- Avantaje
  - putem accesa programatic un element specific - poziția poate fi o variabilă
  - Mai ușor de definit decât repetarea definiției pentru fiecare element
  - Ocupă spațiu continuu și doar strict cât este necesar

Declarația tablourilor

- Un tablou se declară prin specificarea tipului unui singur element, numele tabloului și numărul de elemente

```
tip nume_tablou[nr_elemente]
```

- Numărul de elemente poate fi o constantă (număr sau variabilă fixă) sau o variabilă (standardul C99)
- Exemple,

```
int a[5];

int n = 5;
double matrix[n][n]; //doar cu C99
```

Accesare elemente

- Un tablou declarat cu *n* elemente alocă spațiu de memorie pentru *n* variabile de tipul specificat
- Variabilele ocupă zone de memorie consecutive
- Un element se accesează prin sintaxa

```
nume_tablou[pozitie]
```

- Se folosește indexare de la 0
  - primul element se află la poziția (indexul) 0
  - ultimul element se află la poziția (indexul) *nr_elemente-1*
- Atenție!
  - elementul de la poziția *nr_elemente* nu face parte din tablou
  - accesarea acestei zone sau alte poziții în afara intervalului [0, n-1] poate duce la o eroare la rulare dar acest comportament nu este deterministic

Inițializare tablouri

- Un tablou se poate inițializa folosind inițializatorul { }
- Se dau elementele începând de la cel pe poziția 0
- Pot lipsi elemente, în acest caz restul elementelor se inițializează cu 0
- Dacă nu includem inițializarea atunci valorile din tablou sunt aleatoare
- Se poate inițializa un tablou fără dimensiune
  - se determină automat dimensiunea din numărul de elemente
- Exemple

```
int a[5] = {1, 2, 0};
// a[0] = 1, a[1] = 2, a[2] = a[3] = a[4] = 0
double b[] = {1.2, -0.5};
//dimensiunea lui b este automat determinată ca 2 elemente
```

Inițializare tablouri - avansat

Putem specifica valorile ale doar câtorva elemente specifice

- In loc de

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

- Putem folosi

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- Sau

```
int a[15] = {[14] = 48, [2] = 29, [9] = 7};
```

- Se poate mixa cu iniţializarea clasică

```
int c[] = {0, 1, 2, [4] = 4, [3] = 3};
```

- Se deduce dimensiunea lui c în mod automat ca fiind 5
- Pentru tablouri bidimensionale

```
int I[][] = {[0][0] = 1, [1][1] = 1};
```

Tablouri ca parametri și argumente la funcţii

- Tablou unidimensional
  - În prototipul funcţiei se specifică prin

`tip nume[] sau tip nume[dimensiune]`

  - La apel se trimite numele tabloului
  - Dacă dimensiunea nu este constantă trebuie trimisă separat ca un alt parametru și înaintea tabloului
- Tablou multidimensional
  - În prototipul funcţiei se specifică prin:

`tip nume[][dim_2]...[dim_n]`

  - Trebuie să furnizăm toate dimensiunile în afară de prima (opţională)
  - La apel se trimite numele tabloului
  - Dacă dimensiunile nu sunt constante trebuie trimise separat și ca parametri înaintea tabloului

Tablouri 1D - exemplu simplu

```c
#include <stdio.h>

int main()
{
    int n;
    printf("n = ");
    scanf("%d", &n);
    int a[n];
    for(int i=0; i<n; i++){
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }
    for(int i=0; i<n; i++)
        a[i] = 2*a[i];
    for(int i=0; i<n; i++)
        printf("a[%d] = %d\n", i, a[i]);
    return 0;
}
```

- Exemplu de citire, manipulare și afișare tablou
- declarăm *a* după ce știm dimensiunea
- la scanf tot trebuie să precedăm variabila cu operatorul &

Dimensiunea unui tablou

- Există instrucțiunea sizeof care determină dimensiunea unei variabile sau unui tip în octeți (număr de bytes)
- Se comportă ca o funcție - se apelează cu variabila sau tipul
- Exemplu,

```
int x = 5;
printf("%d %d", sizeof(int), sizeof(x));
```

  - afișează 4 și 4 fiindcă tipul int sau o variabilă de tip int ocupă 4 octeți
- Pentru tablouri se determină dimensiunea totală
- Numărul de elemente se poate găsi prin împărțire cu dimensiunea unui singur element

```
int a[5];
printf("%d", sizeof(a) / sizeof(a[0]));
```

  - afișează 5, sizeof(a) este 20 și sizeof(int) este 4

Tablouri 2D - exemplu complex

```c
#include <stdio.h>

void citire(int n, float a[][n]){
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            printf("a[%d][%d] = ", i, j);
            scanf("%f", &a[i][j]);
        }
    }
}

void diag(int n, float a[][n], float d[]){
    printf("%d\n", sizeof(d));    //4
    for(int i=0; i<n; i++)
        d[i] = a[i][i];

}
```

- funcție pentru citirea elementelor unui tablou 2D de float-uri
- trimitem dimensiunea variabilă și apoi tabloul 2D cu prima dimensiune opțională
- la citire precedăm variabila cu &

```c
int main(){
    int n=2;
    float a[n][n];
    float d[n];
    citire(n, a);
    diag(n, a, d);
    for(int i=0; i<n; i++)
        printf("%f ", d[i]);
    return 0;
}
```

sizeof aplicat pe argumentele funcțiilor nu returnează dimensiunea variabilei originale

Depășirea stivei 2

```
int f(int n){
    int a[n];
}

int main(){
    f(1000000);  //A
    return 0;
}
```

- funcție nerecursivă
- variabila locală *a* este alocată pe stiva de execuție

- rândul unde trebuie revenit după apel

f apel - n = 1000000, a[] local
main - revenire la A

Stiva de execuție pentru *n* = 1000000

- f este apelat la rândul (adresa) *A* din main
- f alocă spațiu pentru un milion de int-uri pe stivă
- sizeof(a) = 1000000*4 ~ 4MB

Interschimbare - revizitat

```c
#include <stdio.h>
void interschimba(int ab[]){
     int t = ab[0];
     ab[0] = ab[1];
     ab[1] = t;
     printf("in func a = %d, b = %d\n", ab[0], ab[1]);
}

int main(){
     int ab[] = {1, 2};
     printf("inainte a = %d, b = %d\n", ab[0], ab[1]);
     interschimba(ab);
     printf("dupa    a = %d, b = %d\n", ab[0], ab[1]);
     return 0;
}
```

- se trimite tabloul ab
- a = 2 și b = 1
- a = 1 și b = 2
- a = 2 și b = 1
- există un alt mecanism pentru trimiterea tablourilor