Limbaje de Programare
Curs 3 – Iterația. Reprezentare internă. Operatori pe biți

Dr. Casandra Holotescu

Universitatea Politehnica Timișoara

Ce discutăm azi...

1. Iterația
2. Reprezentare internă
3. Operații pe biți

Cicluri

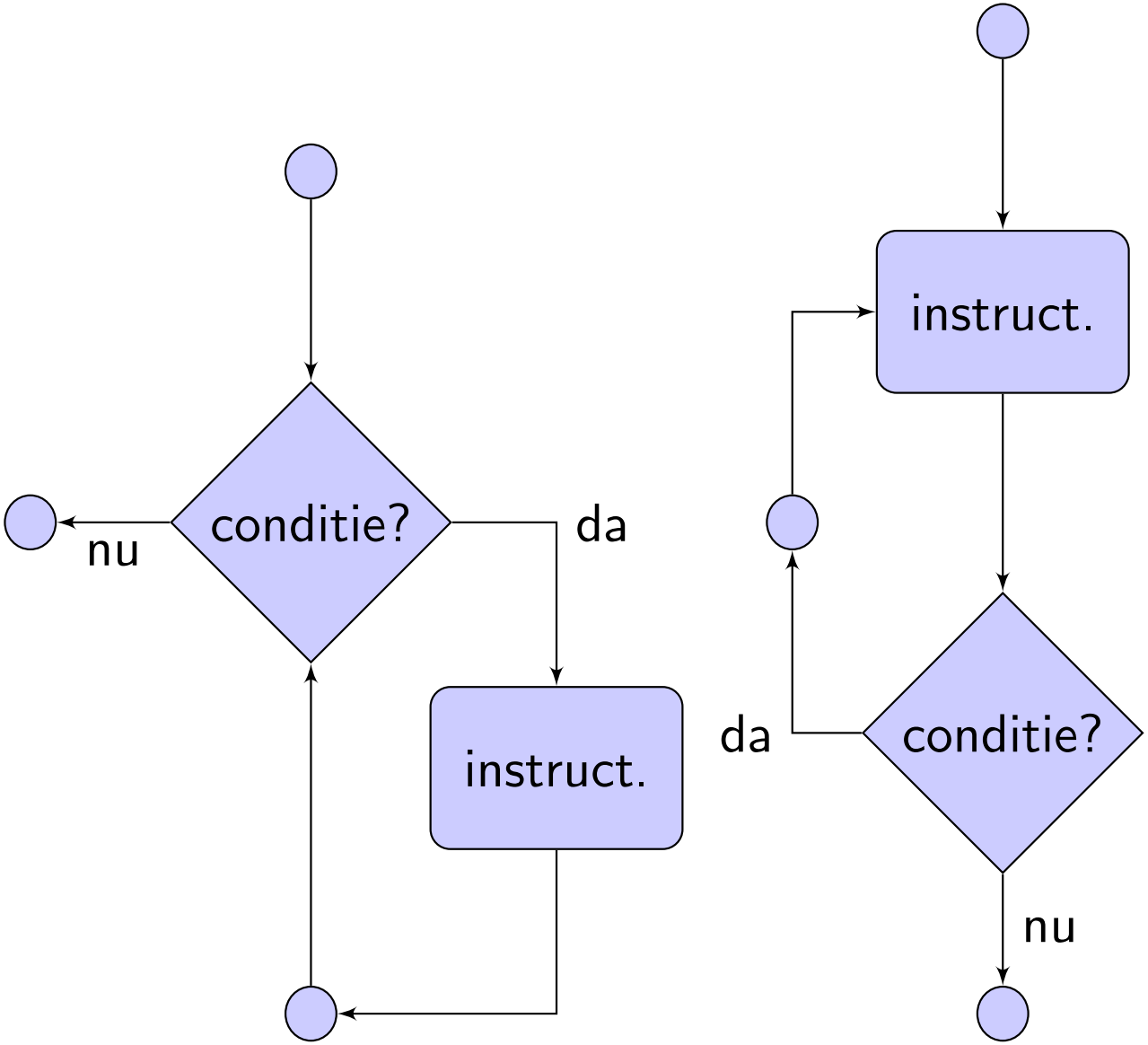
Cum obținem prelucrări repetate ale datelor?

- **recursivitate**: fiecare apel creează noi copii de parametri cu alte valori
- **cicluri**: control direct al iterațiilor, se modifică prin atribuire valorile variabilelor

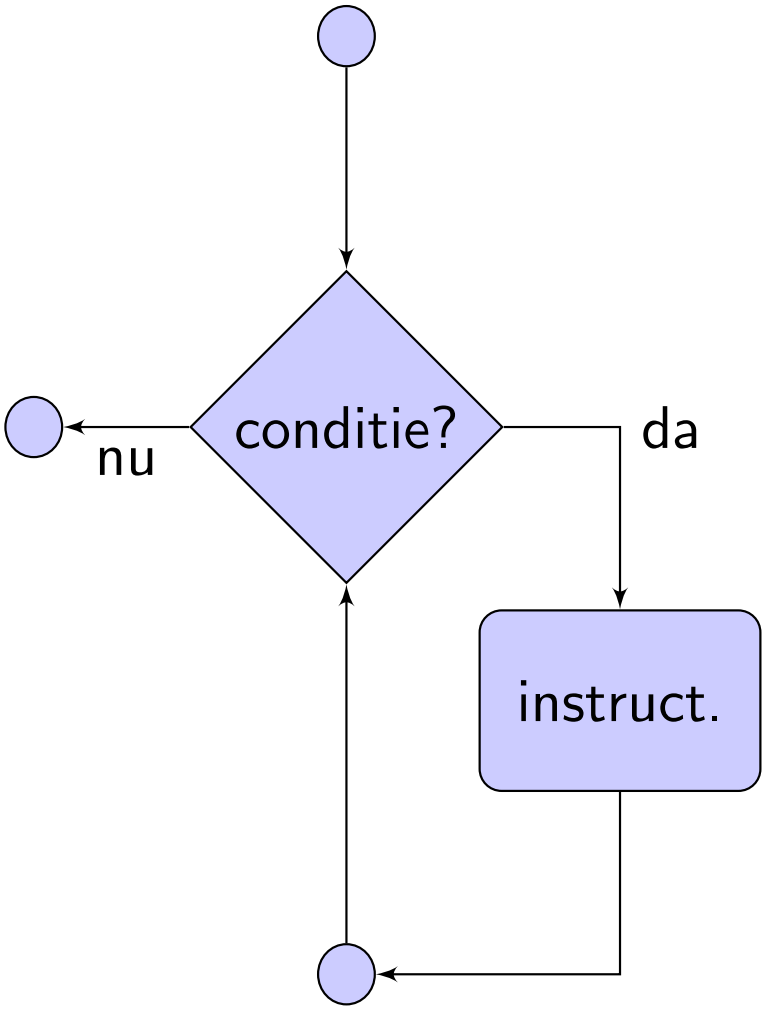
Două tipuri de cicluri:

- cu **test inițial** (condiția anterioară prelucrării)
- cu **test final** (condiția după prelucrare)

Cicluri



Instrucțiunea **while** – test inițial



Instrucțiunea **while**

while (expresie)
instrucțiune

Atenție: parantezele **(** și **)** sunt obligatorii în jurul expresiei!

Semantica instrucțiunii **while**:

- se evaluează expresia
- dacă e adevărată $\Rightarrow$
  - se execută instrucțiunea (sau setul de instrucțiuni)
  - se revine la începutul ciclului, la re-evaluarea expresiei
- dacă e falsă $\Rightarrow$ nu se execută nimic!

$\Rightarrow$ **Corpul ciclului se execută atâta timp cât condiția e adevărată**.

Exemple: citirea unui nr. în mod iterativ

```
#include <ctype.h> // pt. isdigit()
#include <stdio.h> // pt. getchar(), ungetc()

unsigned readnat(void){
    int c; unsigned r = 0;
    while (isdigit(c = getchar()))
    // cat timp e cifra
        r = 10*r + c - '0'; // compune numarul

    ungetc(c, stdin); // pune inapoi ce nu-i cifra
    return r;
}

int main(void) {
      printf("numarul citit: %u\n", readnat());
}
```

Instrucțiunea **for**

O altă formă de ciclu cu test inițial.

for (expr-initializare ; expr-conditie ; expr-actualizare)
instrucțiune

poate fi rescris cu **while** astfel:

expr-initializare ;
while (expr-conditie)
expr-actualizare

Observații:

- oricare din cele 3 expresii (init., cond., act.) poate lipsi, dar cele 2 **;** rămân (putem aveam **for(;;)**)
- daca lipsește *expr-conditie* $\Rightarrow$ ciclu infinit

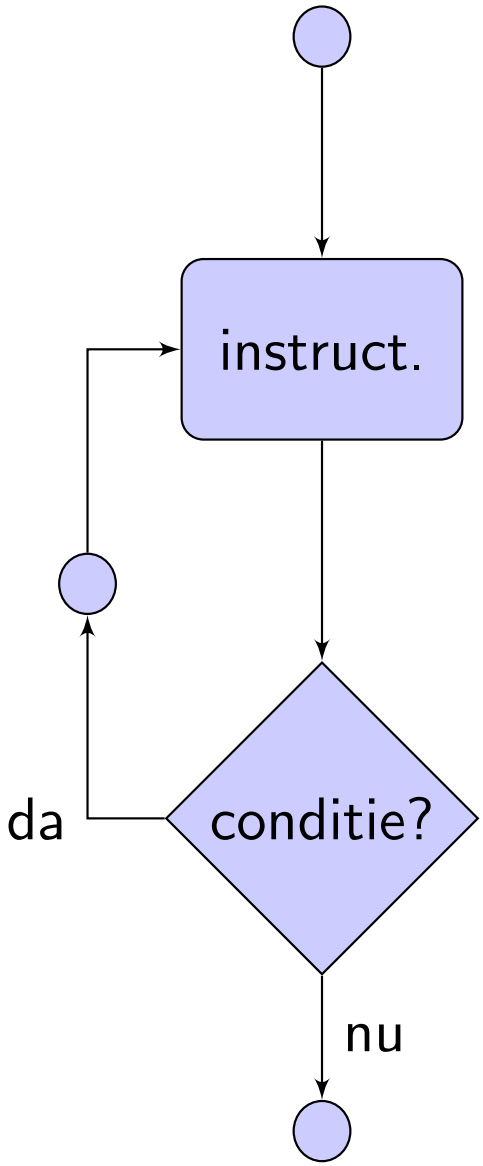
Exemple – suma a *n* cifre de la intrare

```
unsigned sumancif(unsigned n){
    int i=0;
    unsigned r = 0;

    for(i=0; i<n; i++){ // de la 0 la n-1 (n ori)
          int c = getchar(); //citeste caract.

          if(c != EOF && isdigit(c)){ // e cifra?
            r = r + (c-'0'); //aduna la rezultat
        }
    }
    return r;
}
```

Instrucțiunea **do while** – test final



Instrucțiunea **do while**

Dacă știm sigur că un ciclu trebuie executat **cel puțin odată**.

do
    instrucțiune
while (expresie) ;

Semantica instrucțiunii **do while**:

- se execută instrucțiunea (sau setul de instrucțiuni)
- se evaluează expresia
- dacă e adevărată $\Rightarrow$ se revine la începutul ciclului, la re-executarea corpului ciclului
- dacă e falsă $\Rightarrow$ nu se mai execută nimic!

Exemple – cifra max. până la EOF

```
char maxcif(){
    char r = 0; //init cu '\0'

    do{
        int c = getchar(); //citeste caract.

        if(isdigit(c) && c > r){
          // e cifra mai mare?

            r = c; //pastram cifra max.
         }
     } while(c != EOF); //pana la sf. intrarii

    return r;
}
```

Cum gândim ciclurile?

1. identificăm ce variabile se **modifică** în fiecare iterație
2. identificăm **condiția de ieșire** din ciclu
3. avem grijă să **actualizăm** valoarea variabilei/variabilelor pentru a ne apropia de **condiția de ieșire** (altfel ciclul devine **infinit**!)

Instrucțiunea **break**

Produce **ieșirea** din corpul ciclului **imediat înconjurător**.

O folosim dacă nu dorim să continuăm restul prelucrărilor din ciclu.

Sintaxa: **break;**

De regulă, ieșirea din ciclu se face condiționat:

if (conditie) break;

Exemple – numaram cuvintele

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
   int c; unsigned nrw = 0;
   while (1) { //conditie mereu adevarata
      while (isspace(c = getchar()));
       // consuma spatiile
      if (c == EOF)
          break; // gata
      nrw = nrw + 1; // inceput de cuvant
      while (!isspace(c = getchar()) && c != EOF);
         // se citeste cuvantul
    }
   printf("%u\n", nrw);
   return 0;
}
```

Exemple – scriem cuvintele cu majuscula

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
  int c;
  for (;;) { // conditie adevarata
    while (isspace(c = getchar()))
      putchar(c); // citit/scris spatii
    if (c == EOF)  break; // gata
    putchar(toupper(c)); // prima litera
    while ((c = getchar()) != EOF) {
      putchar(c);
      if (isspace(c)) break; // la spatiu iese
    } // si reia ciclul for
  }
  return 0;
}
```

Instrucțiunea **continue**

Produce ieșirea din **iterația curentă** a ciclului **imediat înconjurător** și trecerea la începutul iterației următoare a acestuia, **sărind peste restul instrucțiunilor**.

O folosim dacă nu dorim să continuăm restul prelucrărilor din **iterația curentă**, dar dorim să continuăm ciclul.

Sintaxa: **continue;**

De regulă, saltul la iterația următoare se face condiționat:

if (conditie) continue;

Exemple

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

int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
    //citeste caracter nou, verifica de EOF

        if (!isalpha(c)) //daca nu e litera
            continue; //trece la ciclul urmator

        putchar(c);  //scrie caracterul
        printf(" %d ", c); //si codul sau ASCII
    }
  return 0;
}
```

Recursivitate vs Iterație

Recursivitate:

fiecare apel creează noi copii de parametri cu alte valori
se re-execută aceeași funcție cu alte date

Cicluri:

control direct al iterațiilor
se modifică prin atribuire valorile variabilelor
se reia execuția acelorlași instrucțiuni cu alte valori

Ambele realizează **prelucrări repetate**.

Recursivitatea se poate transforma în iterație și viceversa.

Exemple – Recursivitate și Iterație

```
//factorial recursiv
unsigned int fact_rec(unsigned n, unsigned r){
     if (n==0) return r;
          else return fact_rec(n-1, n*r);
} // se apeleaza fact_rec(n,1);

//factorial iterativ
unsigned int fact_it(unsigned int n){
     unsigned r=1;
     while(n>0){
          r=r*n;
          n=n-1;
     }
     return r;
}
```

Transformarea Recursivității în Iterație

Rescrierea recursivității ca iterație e mai simplă în cazul recursivității cu rezultat parțial (recursivitate la stânga):

- **testul de oprire** al iterației va fi același cu **cazul de bază** al recursivității
- **valoarea inițială** a rezultatului rămâne aceeași
- varianta recursivă: fiecare apel creează **copii noi** de parametri, cu valori proprii (în funcție de cele vechi): ex. n*r, n-1,x*r.
- varianta iterativă: **actualizează** la fiecare iterație valorile variabilelor, după **aceleași relații**. ex. r=n*r, n=n-1, r=x*r.
- ambele variante $\Rightarrow$ **valoarea acumulată** a rezultatului

Exemple – Recursivitate și Iterație

```
//putere recursiva
int pow_rec(int x, unsigned n, int r){
    if (n==0) return r;
        else return pow_rec(x, n-1, x*r);
}

//putere iterativa
int pow_it(int x, unsigned n){
    int r=1;
    while(n>0){
        r=x*r;
        n=n-1;
    }
    return r;
}
```

Reprezentarea obiectelor în memorie

Orice valoare (parametru, variabilă) ocupă loc în memorie.

- **bit** = cea mai mică unitate de memorare, are două valori posibile: 0 sau 1
- **octet** (byte) = grup de 8 biți, cea mai mică unitate care se poate scrie/citi în/din memorie

Operatorul **sizeof** ne dă dimensiunea **în octeți** (bytes) a unui tip sau a unei valori.

Exemplu: *sizeof('A')==sizeof(char)==1*
(un caracter se reprezintă pe 1 octet!)

Dimensiunea tipurilor depinde de sistem (procesor, compilator):
ex. sizeof(int) poate fi 2, 4, 8, ...

Reprezentarea internă – întregi

În memorie, numerele se reprezintă în **binar (baza 2)**, ca șiruri de biți (0 sau 1).

Un întreg fără semn, cu k cifre binare (biți):

$$c_{k-1}c_{k-2}\ldots c_2c_1c_0$$
$$= 2^{k-1} * c_{k-1} + 2^{k-2} * c_{k-2} + \ldots + 2^2 * c_2 + 2 * c_1 + 2^0 * c_0$$
$$= \sum_0^{k-1} 2^i * c_i$$

Unde c_{k-1} e cifra binară (bitul) cea mai semnificativă
iar c_0 cea mai puțin semnificativă.

Ex: $110_{(2)} = 6_{(10)}$ $1100_{(2)} = 12_{(10)}$ $11101_{(2)} = 29_{(10)}$
$11111111_{(2)} = 255_{(10)}$

Reprezentarea internă – întregi

Un întreg cu semn se reprezintă în complement de 2:

bitul superior (primul bit, c_{k-1}) e 1 $\Rightarrow$ nr. e negativ!

$1c_{k-2}...c_2c_1c_0$

$= -2^{k-1} + 2^{k-2} * c_{k-2} + \ldots + 2^2 * c_2 + 2 * c_1 + 2^0 * c_0$

$= -2^{k-1} + \Sigma_0^{k-2} 2^i * c_i$

Exemple (pe 8 biți):

$11111111_{(2)} = -1_{(10)}$ $\quad$ $11111110_{(2)} = -2_{(10)}$

$10000000_{(2)} = -128_{(10)}$

Tipuri întregi

Înainte de **int** putem avea calificatori pentru:

dimensiune: short, long, long long (C99)

semn: unsigned, signed (implicit)

- **char**: signed char [-128, 127] sau unsigned char [0, 255]
- **short**, **int**: $\geq$ 2 octeți, minim $[-2^{15}, 2^{15} - 1]$
- **long**: $\geq$ 4 octeți, minim $[-2^{31}, 2^{31} - 1]$
- **long long**: $\geq$ 8 octeți, minim $[-2^{63}, 2^{63} - 1]$

unsigned are dimensiunea tipului cu semn (b octeti) : $[0, 2^{8b} - 1]$.

$sizeof(short) \leq sizeof(int) \leq sizeof(long) \leq sizeof(long\ long)$

Reprezentarea internă – nr. reale

Numerele reale se reprezintă în memorie aproximativ, în **virgulă flotantă**.

Reprezentare cu: semn, exponent, mantisă.

S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM

float – simplă precizie, 32 de biți:

1 bit de **semn**, 8 biți **exponent**, 23 **mantisă**.

pt. $0 < E < 255$: $\mathbf{nr.} = (-\mathbf{1})^{\mathbf{S}} * \mathbf{2}^{\mathbf{E}-\mathbf{127}} * \mathbf{1.M}$

în rest: 0, numere f. mici, $\pm\infty$

double – dublă precizie, 64 de biți:

1 bit de **semn**, 11 biți **exponent**, 52 **mantisă**.

Operatori pe biți

Oferă acces la reprezentarea binară a datelor în memorie.
Se pot folosi **doar** pentru operanzi de **tip întreg**.

Operator	**Semnificație**
&	ȘI pe biți: 1 & 1 e 1, altfel 0
|	SAU pe biți: 0 | 0 e 0, altfel 1
$\wedge$	SAU EXCLUSIV pe biți: 1 dacă biți diferiti, altfel 0
$\sim$	COMPLEMENT pt biți: 1 devine 0, 0 devine 1
<<	SHIFT (deplasare) la stânga: introduce biți de 0
>>	SHIFT (deplasare) la dreapta: introduce biți de semn

Toți operatorii lucrează simultan pe toți biții operanzilor!

Exemple: operații pe biți

ȘI	SAU	SAU EXCLUSIV
$10110101\ \&$	$10110101\ \vert$	$10110101\ \hat{}$
01110001	01110001	01110001
$= 00110001$	$= 11110101$	$= 11000100$

Complement pe biți:

$\sim 10110101 = 01001010 \qquad \sim 01110001 = 10001110$

Deplasare la stânga:

$10110101 << 1 = 01101010 \qquad 10110101 << 2 = 11010100$

$10110101 << 3 = 10101000 \qquad 10110101 << 6 = 01000000$

Deplasare la dreapta:

$10110101 >> 1 = 01011010 \qquad 10110101 >> 2 = 00101101$

$10110101 >> 3 = 00010110 \qquad 10110101 >> 6 = 00000010$

Exemple

$n << k$ e echivalent cu $n \cdot 2^k$
$n >> k$ e echivalent cu $n/2^k$ (pt. n unsigned!!)

$1 << k$ e 2^k, are doar bitul k pe 1 pt. k $<$ 8*sizeof(int)
$\sim (1 << k)$ are doar bitul k pe 0, restul pe 1
~ 0 are toți biții pe 1 (iar 0 toți pe 0)
$\sim 0 << k$ are k biți din dreapta 0, restul pe 1
$\sim (\sim 0 << k)$ are k biți din dreapta 1, restul pe 0
$\sim (\sim 0 << k) << p$ are k biți pe 1, începând de la bitul p, restul 0

(Atenție, complementul păstrează semnul tipului.)

Exemple

$b \,\&\, 1$ păstrează valoarea bitului b
$b \,\&\, 0$ e 0
$n \,\&\, (1 << k)$ e $! = 0$ dacă bitul k al lui n e 1
$n \,\&\, \sim (1 << k)$ va avea 0 pe bitul k al lui n

$b \mid 1$ e 1
$b \mid 0$ păstrează valoarea bitului b
$n \mid (1 << k)$ va avea 1 pe bitul k al lui n