



# PROGRAMARE PROCEDURALĂ

Bogdan Alexe

[bogdan.alexe@fmi.unibuc.ro](mailto:bogdan.alexe@fmi.unibuc.ro)

Secția Informatică, anul I,  
2016-2017

Cursul 4

# InfO'Clock 2016-2017

- marți, 18-20, amfiteatrul Haret
- la nevoie e disponibilă și sala 204  
(laborator cu 24 calculatoare), miercuri,  
16-18

# Recapitulare – cursul trecut

1. Tipuri de date fundamentale
2. Variabile și constante
3. Expresii și operatori

# Programa cursului

## □ Introducere

- Algoritmi.
- Limbaje de programare.
- Introducere în limbajul C. Structura unui program C.
- Complexitatea algoritmilor.

## □ Fundamentele limbajului C

- Tipuri de date fundamentale. Variabile. Constante. Operatori. Expresii. Conversii.
- Instrucțiuni de control
- Directive de preprocesare. Macrodefiniții.
- Funcții de citire/scriere.
- Etapele realizării unui program C.

## □ Tipuri deriveate de date

- Tablouri. Siruri de caractere.
- Structuri, uniuni, câmpuri de biți, enumerări.
- Pointeri.

## □ Funcții (1)

- Declarație și definire. Apel. Metode de transmisie a parametrilor.
- Pointeri la funcții.



## □ Tablouri și pointeri

- Legătura dintre tablouri și pointeri
- Aritmetică pointerilor
- Alocarea dinamică a memoriei
- Clase de memorare

## □ Siruri de caractere

- Funcții specifice de manipulare.

## □ Fișiere text și fișiere binare

- Funcții specifice de manipulare.

## □ Structuri de date complexe și autoreferite

- Definire și utilizare

## □ Funcții (2)

- Funcții cu număr variabil de argumente.
- Preluarea argumentelor funcției main din linia de comandă.
- Programare generică.

## □ Recursivitate

# Cursul de azi:

1. Operatori și expresii. Conversii
2. Instructiuni de control
3. Directive de preprocesare. Macrodefiniții
4. Etapele realizării unui program în C

# Evaluarea expresiilor

- **precedență și asociativitatea** operatorilor
  - dacă într-o expresie apar mai mulți operatori, atunci evaluarea expresiei respectă **ordinea de precedență** a operatorilor
  - dacă într-o expresie apar mai mulți operatori de aceeași prioritate, atunci se aplică **regula de asociativitate** a operatorilor

# Operatori

1. Operatori aritmetici
2. Operatorul de atribuire
3. Operatori de incrementare și decrementare
4. Operatori de egalitate, logici și relaționali
5. Operatori pe biți
6. Alți operatori:
  - de acces la elemente unui tablou, de apel de funcție, de adresa,
  - de referențiere, sizeof, de conversie explicită, condițional,
  - virgulă

# Alți operatori

- ❑ operatorul de acces la elementele tabloului [ ] 

```
int a[100];  
a[5] = 10;
```
- ❑ operatorul de apel de funcție (): b = f(a);
- ❑ operatorul **adresă** & și operatorul de **dereferețiere** \*
  - ❑ strâns legat de pointeri (cursurile următoare)

```
int a, *p;           // p este un pointer la int  
p = &a;             // p este pointer la a  
*p = 3;             // valoarea lui a devine 3
```

- ❑ operatorul **sizeof**

```
sizeof(a)           // este numărul de octeți  
// ocupări în memorie de a
```
- ❑ operatorul de conversie explicită: (tip)

```
int a = 1, b = 2;  
float media;  
  
media = ( a + (float)b ) / 2;    // media devine 1.5  
media = ( a + b ) / 2;          // media devine 1.0 - incorect!
```

# Alți operatori

## ❑ operatorul condițional ? :

- ❑ operator ternar
- ❑ similar cu instrucțiunea `if`
- ❑ `expresie1? expresie2 : expresie3`
- ❑ dacă `expresie1` e adevarată, execută `expresie2`, altfel execută `expresie3`

```
int a=3, b=5, max;  
max = a > b ? a : b;  
  
a % 2 ? printf("numar impar") : printf("numar par");
```

## ❑ operatorul virgulă

- ❑ evaluarea secvențială a expresiilor (de la stânga la dreapta)
- ❑ valoarea ultimei expresii din înlănțuire este valoarea expresiei compuse
- ❑ cel mai puțin priorită din lista de precedență

```
int i,n,s;  
  
printf("n=");scanf("%d",&n);  
  
for(i=1,s=0;i<=n;s=s+i,i=i+1);
```

# Ordinea de precedență și asociativitate

Operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^=  =	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

# Conversii implicite

conversii.c

```
1 #include <stdio.h>
2
3 int main(){
4
5     int i;
6     i = 1.6 + 1 + 1.7;
7     printf("i = %d \n", i);
8     i = (int)1.6 + 1 + (int)1.7;
9     printf("i = %d \n", i);
10
11    char a = 30, b = 40, c = 10;
12    char d = (a*b)/c;
13    printf("%d \n",d);
14
15    unsigned int ui_one = 1;
16    int i_one = 1;
17    short sh_minus_one = -1;
18    if(sh_minus_one > ui_one)
19        printf("-1 > 1 \n");
20    if(sh_minus_one < i_one)
21        printf("-1 < 1 \n");
22
23    return 0;
24 }
```

Ce afișează  
programul alăturat?

i = 4

i = 3

120

-1 > 1

-1 < 1

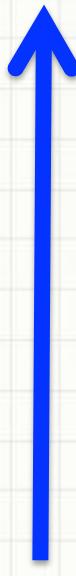
# Conversii implicate

- context
  - este permisă combinarea mai multor operanzi de tipuri diferite într-o singură expresie
- problema
  - operatorii binari, care se aplică asupra a doi operanzi, cer ca **tipul operanzilor să fie același** pentru a putea efectua operația
- soluție: conversia implicită
  - compilatorul **convertește valorile operanzilor la același tip într-un mod transparent** programatorului înaintea generării codului mașină
  - există reguli de conversie implicită
- alternativă
  - conversii explicite: (tip)

# Conversii implicate - reguli

- ❑ când apar într-o expresie tipurile de date **char** și **short** (atât signed și unsigned) sunt convertite la tipul int (**promovarea întregilor**)
- ❑ în orice operație între două operanzi, ambii operanzi sunt convertiți la tipul de date cel mai înalt în ierarhie
- ❑ ierarhia tipurilor de date:  
**(nu există char și short)**
  - ❑ tipul care se reprezintă pe un **număr mai mare de octeți** are un rang mai mare în ierarhie
  - ❑ pentru același tip, varianta **fără semn** are rang mai mare decât cea cu semn
  - ❑ tipurile **reale** au rang mai mare decât tipurile întregi

long double  
double  
float  
unsigned long long int  
long long int  
unsigned long int  
long int  
unsigned int  
int



# Conversii implicite - reguli

## □ conversii implicite la atribuire

- valoarea expresiei din dreapta se convertește la tipul expresiei din stânga
  - pot apărea pierderi – dacă tipul nu este suficient de încăpător

```
char c = 'a';
short sh = 140;
int a = 3, b;
unsigned int u = 1234567u;
long i = 300L;
float f = 80.13f;
double d = 5.75, g;
```

```
b = a + sh;    // val. lui sh convertita la int
a = sh - c;    // val. lui sh si c convertite la int
g = d + f;    // val. lui f convertita la double
f = i + u;    // cal lui u convertita la long
              // rezultatul convertit la float
```

# Conversii implicite

conversii2.c

```
1 #include <stdio.h>
2
3 int main() {
4
5     char ch;
6     int i;
7     float fl;
8     fl = i = ch = 'C';
9     printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);
10    ch = ch + 1;
11    i = fl + 2 * ch;
12    fl = 2.0 * ch + i;
13    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);
14    ch = 1107;
15    printf("Acum ch = %c \n", ch);
16    ch = 80.89;
17    printf("Acum ch = %c \n", ch);
18
19    return 0;
20 }
```

Ce afișează  
programul alăturat?

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Acum ch = S
Acum ch = P
```

# Cursul de azi:

1. Operatori și expresii. Conversii
2. Instrucțiuni de control
3. Directive de preprocesare. Macrodefiniții
4. Etapele realizării unui program în C

# Instrucțiuni de control

- ❑ reprezintă:
  - ❑ elementele fundamentale ale funcțiilor
  - ❑ comenzi date calculatorului
  - ❑ determină fluxul de control al programului (ordinea de execuție a operațiilor din program)
- ❑ **instructiuni de bază**
  - ❑ instructiunea expresie
  - ❑ instructiunea vidă
  - ❑ instructiuni sevențiale/liniare
  - ❑ instructiuni decizionale/selective simple sau multiple
  - ❑ instructiuni repetitive/ciclice/iterative
  - ❑ instructiuni de salt condiționat/necondiționat
  - ❑ instructiunea return

# Instructiuni de control

- ❑ **instructiuni compuse**
  - ❑ create prin combinarea instructiunilor de bază
- ❑ **programare structurată**
  - ❑ Teorema Böhm-Jacopini: fluxul de control poate fi exprimat folosind doar trei tipuri de **instructiuni de control**:
    - ❑ instructiuni secentiale
    - ❑ instructiuni decizionale
    - ❑ instructiuni repetitive

# Instrucțiunea expresie

- ❑ formată dintr-o expresie urmată de semnul ;
  - ❑ expresie;
- ❑ cele mai frecvente
  - ❑ se bazează pe expresii de atribuire, aritmetice și de incrementare / decrementare
    - ❑ adică expresii care au efecte secundare: schimbă valoarea unui operand

Exemple:

```
a = 123;  
b = a + 5;  
b++;
```

- ❑ expresie vs. instrucțiune

**Expresie**

i++

a=a-5

**Instrucțiune**

i++;

a=a-5;

# Instrucțiunea vidă

- ❑ o instrucțiune care constă doar din caracterul ;
  - ❑ folosită în locurile în care limbajul impune existența unei instrucțiuni, dar programul nu trebuie să execute nimic
- ❑ cel mai adesea instrucțiunea vidă apare în combinație cu instrucțiunile repetitive
  - ❑ vezi instrucțiunea for

# Instrucțiunea compusă

- ❑ numită și instrucțiune bloc
- ❑ alcătuită prin gruparea mai multor instrucțiuni și declarații
  - ❑ folosite în locurile în care sintaxa limbajului presupune o singură instrucțiune, dar programul trebuie să efectueze mai multe instrucțiuni
  - ❑ gruparea
    - ❑ includerea instrucțiunilor între accolade, {}
    - ❑ astfel compilatorul va trata secvența de instrucțiuni ca pe o singură instrucțiune
    - ❑ *{secvență de declarații și instrucțiuni }*

# Instrucțiuni decizionale/selective

- ❑ ramifică fluxul de control în funcție de valoarea de adevăr a expresiei evaluate
  
- ❑ limbajul C furnizează două instrucțiuni decizionale
  - ❑ instrucțiunea **if** – instrucțiune decizională simplă
  - ❑ instrucțiunea **switch** - instrucțiune decizională multiplă

# Instrucțiunea IF

- ❑ instrucțiunea selectivă fundamentală
  - ❑ permite selectarea uneia dintre două alternative în funcție de valoarea de adevăr a expresiei testate
- ❑ forma generală:

```
if (expresie)
    {bloc de instructiuni 1};
else
    {bloc de instructiuni 2};
```
- ❑ valoarea expresiei incluse între paranteze rotunde trebuie să fie un scalar
  - ❑ dacă e nenulă se execută *blocul de instrucțiuni 1 (instrucțiunea compusă)*, altfel se execută *blocul de instrucțiuni 2*
- ❑ ramura **else** poate lipsi

# Instrucțiunea IF

- se citesc numerele naturale a și b de la tastatură. Să se afișeze ultima cifră a numărului  $a^b$ .

```
seminar1.c X
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     int a,b;
7     scanf("%d %d",&a,&b);
8     if (a==0)
9     {
10         if (b==0)
11         {
12             printf("0 la puterea 0, caz de nedeterminare \n");
13             return 0;
14         }
15     }
16
17     if(b==0)
18     {
19         printf("1\n");
20         return 0;
21     }
22
23     printf("%d \n", (int)pow(a%10,b%4+4) % 10);
24
25     return 0;
26 }
27
```

# Instructiunea IF

- erori frecvente:
  - nu includem acoladele

```
if (a>b)
    a = a+b;
    b = a;
```

- Întotdeauna se va executa instructiunea `b=a`;
- confundarea operatorul de egalitate `==`, cu operatorul de atribuire `=`

Exemple comparative:

```
a = 2;
if ( a == 10 )
    printf("a este 10 \n");
```

```
a = 2;
if ( a = 10 )
    printf("a este 10 \n");
```

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>□ mesajul <code>a este 10</code> nu va fi afisat<ul style="list-style-type: none"><li>□ după testarea egalității folosind operatorul <code>==</code> se returnează 0<ul style="list-style-type: none"><li>□ (2 nefiind egal cu 10)</li></ul></li></ul></li></ul> | <ul style="list-style-type: none"><li>□ mesajul <code>a este 10</code> va fi afisat întotdeauna<ul style="list-style-type: none"><li>□ expresia <code>a = 10</code><ul style="list-style-type: none"><li>□ a ia valoarea 10</li><li>□ se evaluatează adevărat la executarea instructiunii <code>printf</code></li></ul></li></ul></li></ul> |
|--|---|

# Instrucțiunea IF

- ❑ instrucțiuni IF imbricate
  - ❑ pe oricare ramură poate conține alte instrucțiuni if
- ❑ forma generală:

```
if (expresie1)
    if (expresie2) {bloc de instructiuni 1};
    else {bloc de instructiuni 2};
else
    {bloc de instructiuni 2};
```

Exemplu:

```
int a, b;
// ...
if ( a <= b )
    if ( a == b )
        printf("a = b");
    else
        printf("a < b");
else
    printf("a > b");
```

# Instructiunea IF

- instructiuni IF în cascadă
  - testează succesiv mai multe condiții implementând o variantă de selecție multiplă
- forma generală:

```
if (expresie1) {bloc de instructiuni 1};  
else if (expresie2) {bloc de instructiuni 2};  
else if (expresie3) {bloc de instructiuni 3};  
...  
else {bloc de instructiuni N};
```

```
#include <stdio.h>  
  
int main() {  
    float nota;  
  
    printf("Introduceti o nota in intervalul [1, 10]: ");  
    scanf("%f", &nota);  
  
    if ( nota > 9 && nota <= 10)  
        printf("Calificativul este: EXCELENȚA");  
    else if ( nota > 8 && nota <= 9)  
        printf("Calificativul este: Foarte bine");  
    else if ( nota > 7 && nota <= 8)  
        printf("Calificativul este: Bine");  
    else if ( nota > 5 && nota <= 7)  
        printf("Calificativul este: Suficient");  
    else printf("Calificativul este: Insuficient");  
  
    return 0;  
}
```

# Instrucțiunea SWITCH

- ❑ efectuează selecția multiplă
  - ❑ util când expresia de evaluat are mai multe valori posibile
- ❑ forma generală

```
switch (expresie){  
    case val_const_1: {bloc de instructiuni 1};  
    case val_const_2: {bloc de instructiuni 2};  
    ....  
    case val_const_n: {bloc de instructiuni N};  
    default: {bloc de instructiuni D};  
}
```

# Instrucțiunea SWITCH

- ❑ poate fi întotdeauna reprezentată prin instrucțiunea IF
  - ❑ de regulă prin instrucțiuni IF cascade
- ❑ în cazul instrucțiunii switch fluxul de control sare direct la instrucțiunea corespunzătoare valorii expresiei testate
- ❑ switch este mai rapid și codul rezultat mai ușor de înțeles

# Instructiunea SWITCH

```
#include <stdio.h>

int main()
{
    int nr1, nr2, rez;
    char op;

    printf("Introduceti o expresie aritmetica sub forma: nr1 operator nr2: ");
    scanf("%d %c %d", &nr1, &op, &nr2);

    switch (op)
    {
        case '+': rez = nr1 + nr2; break;
        case '-': rez = nr1 - nr2; break;
        case '*': rez = nr1 * nr2; break;
        case '/': rez = nr1 / nr2; break;
        case '%': rez = nr1 % nr2; break;
    }

    printf("Valoarea expresiei aritmetice introduse este: %d \n", rez);

    return 0;
}
```

Rezultatul unei rulări a acestui program este:

```
Introduceti o expresie aritmetica sub forma: nr1 operator nr2: 4 + 7
Valoarea expresiei aritmetice introduse este: 11
```

# Instrucțiunea SWITCH

- se citesc numerele naturale a și b de la tastatură. Să se afișeze ultima cifră a numărului  $a^b$ .

The screenshot shows a code editor window titled "seminar1\_2.c". The code is written in C and calculates the last digit of  $a^b$ . It includes #include directives for stdio.h and math.h, defines main(), and uses scanf to read integers a and b. It handles cases where a or b are zero. A switch statement is used to determine the last digit based on the value of b modulo 4.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      int a,b;
7      scanf("%d %d",&a,&b);
8      if (a==0)
9      {
10         if (b==0)
11         {
12             printf("0 la puterea 0, caz de nedeterminare \n");
13             return 0;
14         }
15     }
16
17     if(b==0)
18     {
19         printf("1\n");
20         return 0;
21     }
22
23     a = a%10;
24     switch (b%4)
25     {
26         case 0: printf("%d\n",a*a*a*a % 10); break;
27         case 1: printf("%d\n",a); break;
28         case 2: printf("%d\n",a*a % 10); break;
29         case 3: printf("%d\n",a*a*a % 10); break,
30     }
31     return 0;
32 }
```

# Instrucțiunea SWITCH

- ❑ efectuează selecția multiplă
  - ❑ util când expresia de evaluat are mai multe valori posibile
- ❑ forma generală

```
switch (expresie){  
    case val_const_1: bloc de instructiuni 1;  
    case val_const_2: bloc de instructiuni 2;  
    ....  
    case val_const_n: bloc de instructiuni N;  
    default: bloc de instructiuni D;  
}
```

# Instrucțiunea SWITCH

- ❑ mod de funcționare și constrângeri:
  - ❑ expresie se evaluează o singură dată la intrarea în instrucțiunea switch
  - ❑ expresie trebuie să rezulte într-o valoare întreagă (poate fi inclusiv caracter, dar nu valori reale sau siruri de caractere)
  - ❑ valorile din ramurile **case** notate **val\_ const\_i** (numite și etichete) trebuie să fie constante întregi (sau caracter), reprezentând o singură valoare
  - ❑ nu se poate reprezenta un interval de valori
  - ❑ instrucțiunile care urmează după etichetele **case** nu trebuie incluse între accolade, deși pot fi mai multe instrucțiuni, iar ultima instrucțiune este de regulă instrucțiunea **break**

# Instrucțiunea SWITCH

- mod de funcționare și constrângeri:
  - dacă valoarea expresiei se potrivește cu vreuna din valorile constante din ramurile case, atunci se vor executa instrucțiunile corespunzătoare acelei ramuri, altfel se execută instrucțiunea de pe ramura **default** (dacă există)
  - dacă nu s-a întâlnit **break** la finalul instrucțiunilor de pe ramura pe care s-a intrat, atunci se continuă execuția instrucțiunilor de pe ramurile consecutive (fără verificarea etichetei) până când se ajunge la **break** sau la sfârșitul instrucțiunii **switch**, moment în care se ieșe din instrucțiunea **switch** și se trece la execuția instrucțiunii imediat următoare
  - ramura **default** este opțională iar poziția relativă a acesteia printre celelalte ramuri nu este relevantă
  - dacă nici o etichetă nu se potrivește cu valoarea expresiei testate și nu există ramura **default**, atunci instrucțiunea **switch** nu are nici un efect

# Instrucțiunea SWITCH

- omiterea instrucțiunii break de la finalul unei ramuri case
  - accidentală - este o eroare frecventă
  - deliberată - permite fluxului de execuție să intre și pe ramura case următoare

```
...
switch (luna)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: nr_zile = 31;
               break;
    case 4:
    case 6:
    case 9:
    case 11: nr_zile = 30;
               break;
    case 2: if (bisect == 1) nr_zile = 29;
              else nr_zile = 28;
              break;
    default: printf("Luna trebuie sa fie in intervalul [1, 12] !");
}
```

# Instructiunea SWITCH

seminar1\_2.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     int a,b;
7     scanf("%d %d",&a,&b);
8     if (a==0)
9     {
10         if (b==0)
11         {
12             printf("0 la puterea 0, caz de nedeterminare \n");
13             return 0;
14         }
15     }
16
17     if(b==0)
18     {
19         printf("1\n");
20         return 0;
21     }
22
23     a = a%10;
24     switch (b%4)
25     {
26         case 0: printf("%d\n",a*a*a*a % 10);
27         case 1: printf("%d\n",a);
28         case 2: printf("%d\n",a*a % 10);
29         case 3: printf("%d\n",a*a*a % 10);
30     }
31     return 0;
32 }
```

12 33  
2  
4  
8

# Instrucțiuni repetitive

- ❑ sunt numite și instrucțiuni iterative sau ciclice
- ❑ efectuează o serie de instrucțiuni în mod repetat fiind condiționate de o expresie de control care este evaluată la fiecare iterare
- ❑ instrucțiunile iterative furnizate de limbajul C sunt:
  - ❑ instrucțiunea repetitivă cu testare inițială **while**
  - ❑ instrucțiunea repetitivă cu testare finală **do-while**
  - ❑ instrucțiunea repetitivă cu testare inițială **for**

# Instrucțiunea WHILE

- execută în mod repetat o instrucțiune atât timp cât expresia de control este evaluată la adevărat
- evaluarea se efectuează la începutul instrucțiunii
  - dacă rezultatul corespunde valorii logice adevărat
    - se execută corpului instrucțiunii, după care se revine la testarea expresiei de control
  - acești pași se repetă până când expresia va fi evaluată la fals
    - acesta va determina ieșirea din instrucțiune și trecerea la instrucțiunea imediat următoare
- forma generală: **while** (*expresie*) {bloc de instrucțiuni}

# Instructiunea WHILE

A programmer heads out to  
the store. His wife says  
"while you're out, get some  
milk."

He never came home.

# Instrucțiunea WHILE

```
sumaNumere.c ✘
1 #include <stdio.h>
2
3 int main()
4 {
5     int nr, i , suma;
6     printf("Introduceti un numar intreg: ");
7     scanf("%d",&nr);
8
9     i = 0; suma = 0;
10    while (i<=nr)
11    {
12        suma += i;
13        i++;
14    }
15    printf("Suma numerelor mai mici sau egale decat %d este: %d\n", nr, suma);
16
17    return 0;
18
19 }
20
```

## Observații

- valorile care participă în expresia de control să fie inițializate înainte
- evitare ciclului infinit

# Instrucțiunea WHILE

sumaNumere.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int nr, i , suma;
6     printf("Introduceti un numar intreg: ");
7     scanf("%d",&nr);
8
9     i = 0; suma = 0;
10    while (((i=i+1) && (i<=nr) && (suma+i) );
11        printf("Suma numerelor mai mici sau egale decat %d este: %d\n", nr, suma);
12
13    return 0;
14
15 }
16
```

## Observații

- Dacă o expresie nu mai este adevărată nu se mai continuă cu evaluarea expresiilor următoare

# Instrucțiunea WHILE

```
unsigned int i=3;  
while (i>=0){  
    printf("%d\n",i);  
    i--;  
}
```

Ce afișează sevența de cod alăturată?

CICLEAZA!!!

# Instrucțiunea DO-WHILE

- ❑ efectuează în mod repetat o instrucțiune atât timp cât expresia de control este adevărată
- ❑ evaluarea se face la finalul fiecărei iterații
  - ❑ corpul instrucțiunii este executat cel puțin o dată
- ❑ forma generală: **do** {bloc de instrucțiuni} **while** (expresie);
- ❑ eroare frecventă: omiterea caracterului punct și virgulă de la finalul instrucțiunii

# Instructiunea DO-WHILE

```
#include <stdio.h>
#define N 100

int main()
{
    int nr, i, suma;
    int v[N];

    do
    {
        printf("Introduceti numarul de elemente (1 <= nr <= 100): ");
        scanf("%d", &nr);
    } while(nr<1 || nr >100);

    i = 0; suma = 0;
    do
    {
        printf("v[%d]: ", i);
        scanf("%d", &v[i]);
        suma += v[i];
        i++;
    } while (i<nr);

    printf("Suma elementelor vectorului este: %5d\n", suma);

    return 0;
}
```

Rezultatul unei rulări a acestui program este:

```
Introduceti numarul de elemente (1 <= nr <= 100): 5
v[0]: 4
v[1]: 7
v[2]: 2
v[3]: 10
v[4]: 5
Suma elementelor vectorului este:      28
```

# Instrucțiunea FOR

- ❑ evaluarea expresiei de control se face la începutul fiecărei iterății

- ❑ forma generală:

```
for (expresii_init; expresie_control; expresie_ajustare)
    {bloc de instructiuni}
```

- ❑ poate fi întotdeauna transcrisă folosind o instrucțiune while:

```
expresii_init;
while (expresie_control)
    {bloc de instructiuni
        expresii_ajustare;}
```

# Instrucțiunea FOR

```
// insumarea elementelor din vectorul de intregi cu for  
  
for (i = 0, suma = 0; i < nr ; i++)  
{  
    printf("v[%d]: ", i);  
    scanf("%d", &v[i]);  
    suma += v[i];  
}
```

- instrucțiunea for permite ca elementul de ajustare din antetul instrucțiunii să cuprindă mai multe expresii
  - se poate ajunge chiar și la situația în care corpul instrucțiunii nu mai conține nici o instrucțiune de executat
  - se folosește **instrucțiunea vidă** (punct și virgulă) pentru a indica sfârșitul instrucțiunii for

```
// insumarea elementelor din vectorul de intregi cu for  
  
for (i = 0, suma = 0; i < nr ; suma += v[i], i++);  
printf("Suma elementelor este: %d", suma);
```

# Instrucțiunea FOR

```
int i=0;  
for(; i<=5; i++);  
    printf("%d", i);
```

Ce afișează sevența de cod alăturată?

Afișează 6

# Instrucțiunile break, continue și goto

- ❑ realizează **salturi**
  - ❑ îintrerup controlului secvențial al programului și continuă execuția dintr-un alt punct al programului sau chiar provoacă ieșirea din program
- ❑ instrucțiunea **break** provoacă ieșirea din instrucțiunea curentă
- ❑ instrucțiunea **continue** provoacă trecerea la iterată imediat următoare în instrucțiunea repetitivă
- ❑ instrucțiunea **goto** produce un salt la o etichetă predefinită în cadrul aceleasi funcții

# Instrucțiunea goto

- instrucțiunea **goto** produce un salt la o etichetă predefinită în cadrul aceleiași funcții
- forma generală: **goto eticheta**
  - unde eticheta este definită în program
  - eticheta: instructiune

```
int i=0;  
eticheta:  
    if(i%3!=0)  
        printf("i=%d\n",i);  
    i++;  
    if(i<10)  
        goto eticheta;  
return 0;
```

i=1  
i=2  
i=4  
i=5  
i=7  
i=8

# Instrucțiunile break, continue și goto

```
//Insumarea tuturor numerelor prime
dintr-un vector de intregi, pana la
intalnirea primului numar multiplu de
100
#include <stdio.h>
#include <math.h>
int main()
{
    int v[]={640,2,29,1,49,
              33,23,800,47,3};
    int suma=0;
    int i;
    int nr=sizeof(v)/sizeof(int);
    for (i=0; i<nr; i++)
    {
        if (v[i]%100==0)
            goto afisare_suma;
        if (v[i]<2)
            continue;
```

```
        int prim=1;
        int k;
        double epsilon=0.001;
        int limit= (int) (sqrt(v
[i])+epsilon);
        for (k=2; k<=limit; k++)
            if (v[i]%k==0)
                {
                    prim=0;
                    break;
                }
        if (prim)
            suma+=v[i];
    }
afisare_suma:
    printf("Suma este %d", suma);
    return 0;
}
```

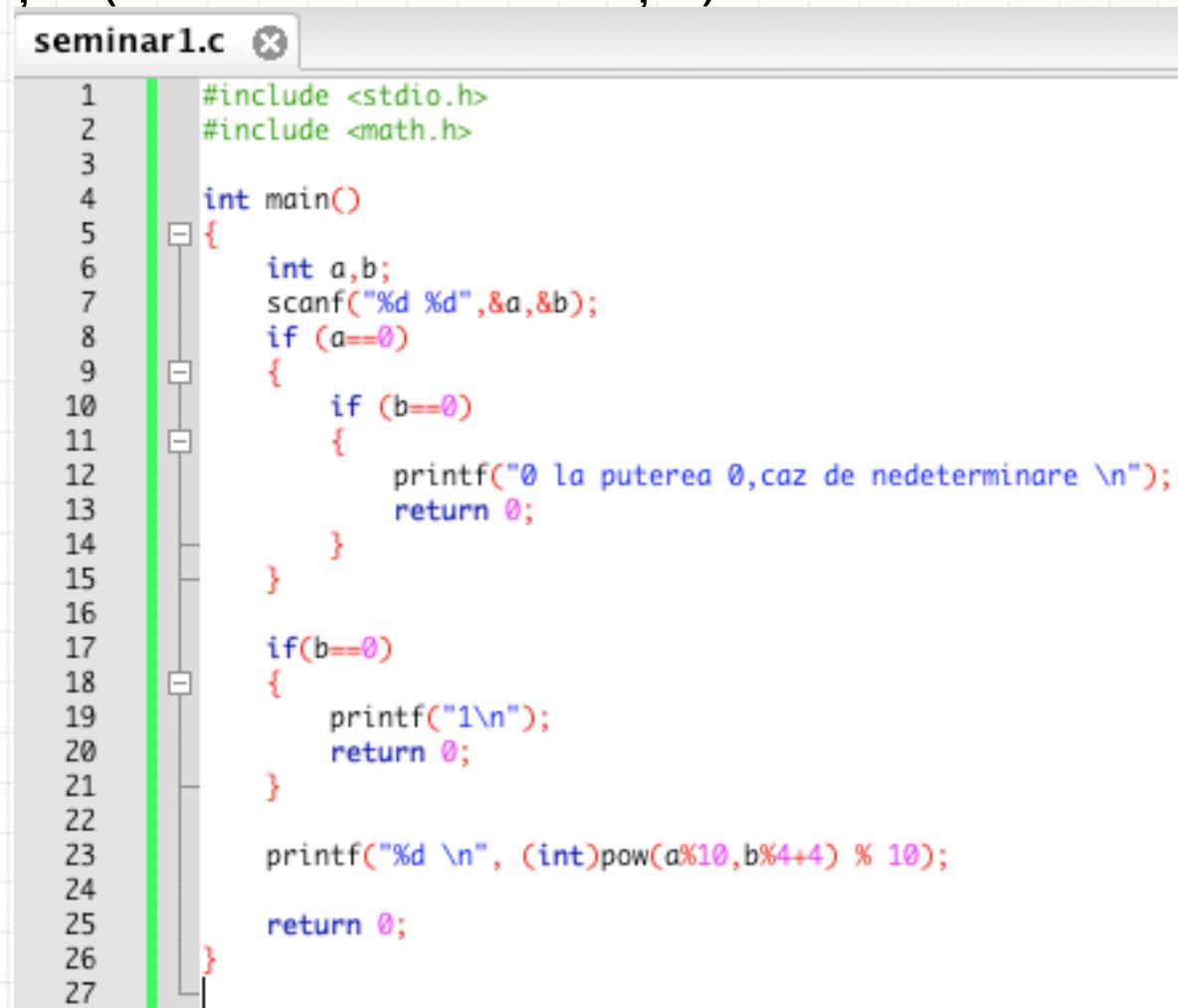
# Instrucțiunile break, continue și goto

```
//Acceasi problema dar fara a
utiliza break, continue si goto
#include <stdio.h>
#include <math.h>
int main()
{
    int v[]={640,2,29,1,49,
              33,23,800,47,3};
    int suma=0;
    int i=0;
    int nr=sizeof(v)/sizeof(int);
    while (i<nr && v[i]%100!=0)
    {
        if (v[i]>=2)
        {
            int prim=1;
            int k=2;
            double epsilon=0.001;
```

```
            int limit= (int) (sqrt(v[i]))
                +epsilon);
            while (prim && k<=limit)
            {
                if (v[i]%k==0)
                    prim=0;
                k++;
            }
            if (prim)
                suma+=v[i];
        } i+
    }
    printf("Suma este %d", suma);
    return 0;
```

# Instrucțiunea RETURN

- se folosește pentru a returna fluxul de control al programului apelant dintr-o funcție (main sau altă funcție)
- are două forme:
  - return;
  - return expresie;



```
seminar1.c
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     int a,b;
7     scanf("%d %d",&a,&b);
8     if (a==0)
9     {
10         if (b==0)
11         {
12             printf("0 la puterea 0, caz de nedeterminare \n");
13             return 0;
14         }
15     }
16     if(b==0)
17     {
18         printf("1\n");
19         return 0;
20     }
21     printf("%d \n", (int)pow(a%10,b%4+4) % 10);
22
23     return 0;
24
25 }
```

# Cursul de azi:

1. Operatori și expresii. Conversii
2. Instructiuni de control
3. Directive de preprocesare. Macrodefiniții
4. Etapele realizării unui program în C

# Preprocesare în limbajul C

- ❑ preprocesarea apare înaintea procesului de compilare
- ❑ constă în substituirea simbolurilor din codul sursă pe baza directivelor de preprocesare
- ❑ directivele de preprocesare sunt precedate de caracterul diez **#**
- ❑ preprocesarea asigură
  - ❑ includerea conținutului fișierelor (de obicei a fișierelor *header*)
  - ❑ definirea de macrouri (macrodefiniții)
  - ❑ compilarea condiționată

# Constante simbolice și macro-uri

- directiva utilizată este **#define**
- definirea unei **constante simbolice** este un caz special al definirii unui macro

```
#define nume text
```

- În timpul preprocesării **nume** este înlocuit cu **text**
- **text** poate să fie mai lungă decât o linie, continuarea se poate face prin caracterul \ pus la sfârșitul liniei
- **text** poate să lipsească, caz în care se definește o constantă vidă

# Constante simbolice și macro-uri

- directiva utilizată este **#define**
- definirea unei **constante simbolice** este un caz special al definirii unui macro
  - #define nume text**
- în timpul preprocesării **nume** este înlocuit cu **text**
- înlocuirea se continuă până în momentul în care **nume** nu mai este definit sau până la sfârșitul fișierului
  - renunțarea la definirea unei constante simbolice se poate face cu directiva **#undef nume**

# Constante simbolice și macro-uri

- ❑ definirea unui **macro**:

```
#define nume (p1, p2, ..., pn) text
```

- ❑ numele macro-ului este nume
- ❑ parametri macro-ului sunt **p1, p2, ..., pn**
- ❑ textul substituit este **text**
- ❑ parametrii formali sunt substituți de cei actuali în text
- ❑ apelul macro-ului este similar apelului unei funcții  
**nume(p\_actual1, p\_actual2, ..., p\_actualn)**

# Constante simbolice și macro-uri

```
#include <stdio.h>

//constante simbolice
#define ALPHA 30
#define BETA ALPHA+10
#define GAMMA (ALPHA+10)

//macro-uri
#define MIN(a,b) (((a)<(b))?(a):(b))
#define ABS1(x) (x<0)?-x:x
#define ABS2(x) (((x)<0)?-(x):(x))
#define INTER(tip,a,b) \
    {tip c; c=a; a=b; b=c;}
```

```
int main()
{
    int x=2*BETA;
    int y=2*GAMMA;
    printf("%d %d\n",x,y); //70 80
    int m=MIN(x,y);
    printf("%d\n",m); //70
    int a=ABS1(x-y);
    int b=ABS2(x-y);
    printf("%d %d\n",a,b); //-150 10
    INTER(int,a,b);
    printf("%d %d\n",a,b); //10 -150
    INTER(int,a,b);
    printf("%d %d\n",a,b); //-150 10
    return 0;
}
```

# Macro-uri

- invocarea unui **macro** presupune înlocuirea apelului cu **textul** macro-ului respectiv
  - se generează astfel instrucțiuni la fiecare invocare și care sunt ulterior compilate
  - se recomandă astfel utilizarea doar pentru calcule simple
  - parametrul formal este înlocuit cu textul corespunzător parametrului actual, corespondența fiind pur pozitională
- timpul de procesare este mai scurt când se utilizează macro-uri (apelul funcției necesită timp suplimentar)

# Compilarea condiționată

- ❑ facilitează dezvoltarea dar în special testarea codului
- ❑ directivele care pot fi utilizate: `#if`, `#ifdef`, `#ifndef`
- ❑ directiva `#if`:

```
#if expr  
    text  
#endif
```

```
#if expr  
    text1  
#else  
    text2  
#endif
```

- ❑ unde `expr` este o expresie constantă care poate fi evaluată de către preprocesor, `text`, `text1`, `text2` sunt porțiuni de cod sursă
- ❑ dacă `expr` nu este zero atunci `text` respectiv `text1` sunt compilate, altfel numai `text2` este compilat și procesarea continuă după `#endif`

# Compilarea condiționată

- ❑ directiva **#ifdef**:

```
#ifdef nume  
    text  
#endif
```

```
#ifdef nume  
    text1  
#else  
    text2  
#endif
```

- ❑ unde **nume** este o constantă care este testată de către preprocesor dacă este definită, **text**, **text1**, **text2** sunt porțiuni de cod sursă
- ❑ dacă **nume** este definită atunci **text** respectiv **text1** sunt compilate, altfel numai **text2** este compilat și procesarea continuă după **#endif**

# Compilarea condiționată

## □ directiva `#ifndef`:

```
#ifndef nume  
    text  
#endif
```

```
#ifndef nume  
    text1  
#else  
    text2  
#endif
```

- unde `nume` este o constantă care este testată de către preprocesor dacă NU este definită, `text`, `text1`, `text2` sunt porțiuni de cod sursă
- dacă `nume` NU este definită atunci `text` respectiv `text1` sunt compilate, altfel numai `text2` este compilat și procesarea continuă după `#endif`

# Compilarea condiționată

- directivele **#ifdef** și **#ifndef** sunt folosite de obicei pentru a evita incluziunea multiplă a modulelor în programarea modulară
- la începutul fiecărui fișier *header* se practică de obicei o astfel de secvență

```
#ifndef _MODUL_H_
#define _MODUL_H_
...
#endif /* _MODUL_H_ */
```

- există o serie de macro-uri predefinite care nu trebuie re/definite:
  - \_DATE\_** data compilării
  - \_CDECL\_** apelul funcției urmărește convențiile C
  - \_STDC\_** definit dacă trebuie respectate strict regulile ANSI C
  - \_FILE\_** numele complet al fișierului curent compilat
  - \_FUNCTION\_** numele funcției curente
  - \_LINE\_** numărul liniei curente

# Compilarea condiționată

```
#include <stdio.h>
//constante simbolice
#define DEBUG
#define X -3
#define Y 5

int main()
{
#ifndef DEBUG
    printf("Suntem in functia %s\n", __FUNCTION__); //main
#endif
#if X+Y
    double a=3.1;
#else
    double a=5.7;
#endif
    a*=2;
#ifndef DEBUG
    printf("La linia %d valoarea lui a este %f\n", __LINE__,a); //18 6.2
#endif
    a+=10;
    printf("a este %f",a); //16.2
    return 0;
}
```

# Cursul de azi:

1. Operatori și expresii. Conversii
2. Instructiuni de control
3. Directive de preprocesare. Macrodefiniții
4. Etapele realizării unui program în C

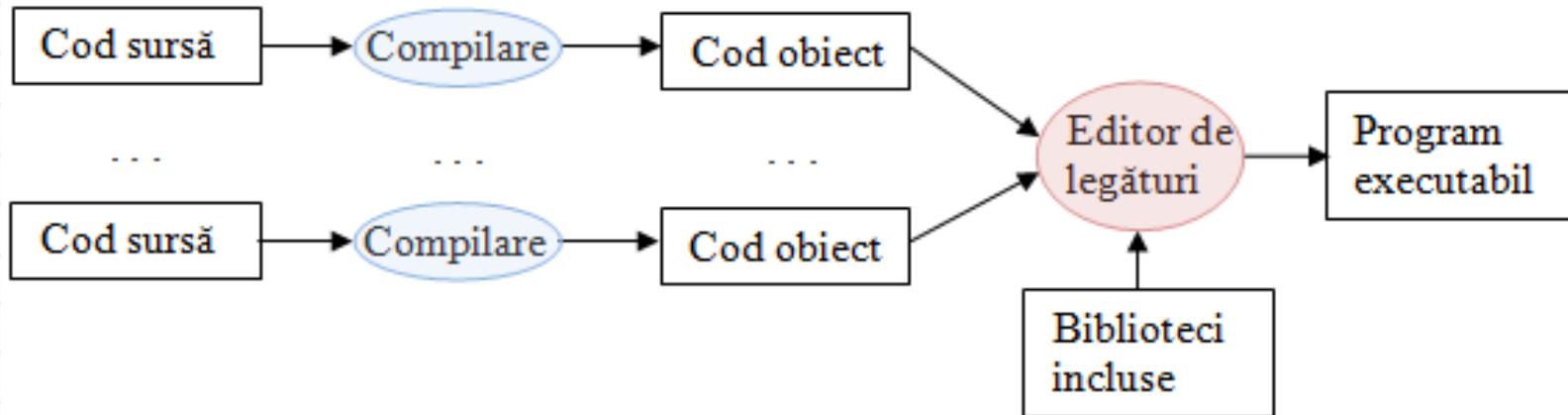
# De la codul sursă la programe executabil

- etape
  - editarea codului sursă
    - salvarea fișierului cu extensia .c
  - preprocesarea
    - efectuarea directivelor de preprocesare
    - ca un editor – modifică și adaugă la codul sursă
  - compilarea
    - verificarea sintaxei
    - transformare în cod obiect (limbaj mașină) cu extensia .obj
      - nu este încă executabil !
  - link-editarea (editarea legăturilor)
    - combinarea codului obiect cu alte coduri obiect (al bibliotecilor asociate fișierelor header)
    - transformarea adreselor simbolice în adrese reale

# De la codul sursă la programe executabile

## □ etape

- editarea codului sursă
- preprocessarea
- compilarea
- link-editarea (editarea legăturilor)



# De la codul sursă la programe executabil

## □ etape

- editarea codului sursă
- preprocessarea
- compilarea
- link-editarea (editarea legăturilor)

