



# PROGRAMARE PROCEDURALĂ

Bogdan Alexe

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

Secția Informatică, anul I,

2018-2019

Cursul 5

# BALUL BOBOCILOR

13 NOIEMBRIE • 22:00

frate||i  
LOUNGE & CLUB

## RISE OF ARTIFICIAL



• Do you believe in myths? •

REZERVĂRI

FAA : 0720471876 | FI : 0721256157 | FMI : 0735187522

# Seminarul 3

- online pe Moodle;
- tema seminarului: structuri, uniuni, enumerări, câmpuri de biți, tipuri definite de utilizator (typedef);
- probleme rezolvate și propuse (eventual mai adaug în weekend alte probleme) ;
- se folosește funcția qsort din biblioteca stdlib.h

# Funcția qsort

- antetul funcției qsort este:

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element,  
int (*cmp) (const void *, const void *))
```

- adresa = pointer la adresa primului element al tabloului ce urmează a fi sortat  
(pointer generic – nu are o aritmetică precizată)
- nr\_elemente = numărul de elemente al vectorului
- dimensiune\_element = dimensiunea în octeți a fiecărui element al tabloului  
(char = 1 octet, int = 4 octeți, etc)
- **cmp – funcția de comparare a două elemente, pointer la o funcție**

# Funcția qsort

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element,  
int (*cmp) (const void *, const void *))  
  
int cmp(const void *a, const void *b)
```

adresele a două elemente din tablou

Cmp este o funcție generică comparator, compară 2 elemente de orice tip din tablou. Întoarce:

- un număr < 0 dacă vrem elementul de la adresa **a** la stânga (înaintea) elementului de la adresa **b**
- un număr > 0 dacă vrem elementul de la adresa **a** la dreapta (după) elementului de la adresa **b**
- 0, dacă nu contează

# Funcția qsort pentru întregi

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element, int  
(*cmp) (const void *, const void *))
```

Exemplu de funcție cmp pentru sortarea unui vector de numere întregi:

```
int cmp(const void* a, const void *b)  
{  
    int va, vb;  
    va = *(int*)a;  
    vb = *(int*)b;  
    if(va < vb) return -1;  
    if(va > vb) return 1;  
    return 0;  
}
```



```
int cmp(const void* a, const void *b)  
{  
    return *(int*)a - *(int*)b;  
}
```

# Funcția qsort pentru întregi

```
exempluqsort.c

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int cmp(const void* a, const void *b)
5 {
6     return *(int*)a - *(int*)b;
7 }
8
9 int main()
10 {
11     int v[] = {0, 5, -6, 9, 7, 12, 8, 7, 4};
12     qsort(v, 9, sizeof(int), cmp);
13     for( int i = 0; i < sizeof(v)/sizeof(int); i++)
14         printf("%d \t", v[i]);
15     printf("\n");
16     return 0;
17 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
```

```
-6      0      4      5      7      7      8      9      12
```

# Funcția qsort pentru structuri

Exercițiu seminar: avem o structură care reține numele, prețul, cantitatea pentru fiecare produs dintr-un magazin. Se dă un vector de asemenea produse pe care vreau să îl sorteze descrescător după preț și în caz de prețuri egale în ordinea alfabetică a numelor produselor.

```
produs.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 struct Produs
6 {
7     char nume[25];
8     double pret;
9     double cantitate;
10 };
11
12
13 int cmpProduse(const void *a, const void* b)
14 {
15     if (((struct Produs *)a)->pret == ((struct Produs *)b)->pret)
16         return strcmp(((struct Produs *) a)->nume,((struct Produs *)b)->nume);
17
18     if (((struct Produs *)a)->pret > ((struct Produs *)b)->pret)
19         return -1;
20     return +1;
21 }
22 }
```

# Recapitulare – cursul trecut

1. Tipuri derivate de date: tablouri, siruri de caractere, structuri, campuri de biti, uniuni, enumerari, tipuri definite de utilizator (typedef).
2. Instructiuni de control.
3. Directive de procesare. Macrodefinitii.

# Programa cursului

## □ Introducere

- Algoritmi
- Limbaje de programare.

## □ Fundamentele limbajului C

- Introducere în limbajul C. Structura unui program C.
- Tipuri de date fundamentale. Variabile. Constante. Operatori. Expresii. Conversii.
- Tipuri derivate de date: tablouri, siruri de caractere, structuri, uniuni, câmpuri de biți, enumerări, pointeri
- Instrucțiuni de control
- Directive de preprocesare. Macrodefiniții.
- Funcții de citire/scriere.
- Etapele realizării unui program C.



## □ Fișiere text

- Funcții specifice de manipulare.

## □ Funcții (1)

- Declarație și definire. Apel. Metode de transmitere 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 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

# Cuprinsul cursului de azi

1. Instrucțiuni de control (break, continue, goto, return).
2. Etapele realizării unui program C.
3. Directive de preprocessare. Macrodefiniții.
4. Funcții de citire/scriere.

# Exemplu FOR

- putem scrie un program într-un singur for;

```
exempluFor1.c

1 #include <stdio.h>
2
3 int main()
4 {
5     int i, j, n;
6     int v[] = {-7, 29, 76, 8};
7     n = sizeof(v)/sizeof(int);
8
9     for(i = 0, j = 1; n > 1 && i < n; (v[i] > v[j]) && (v[i] = v[i] + v[j] - (v[j] = v[i])), 
10        i++, j = j < n-1 ? -i, j+1 : i+1);
11
12    for(i = 0; i < n; i++)
13        printf("%d\t", v[i]);
14
15    printf("\n");
16    return 0;
17 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ gcc exempluFor1.c
```

```
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
```

```
-7      8      29      76
```

# Exemple FOR

exempluFor2.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i, j, n, r, s, w;
6     int v[100];
7
8     for(i = 0, j = 1, r = s = w = 0, printf("Numarul de elemente: "), scanf("%d", &n), printf("\nElementele:\n");
9         i < n ? 1 : (r == 0 ? (i = 0, r = 1) : (s == 0 ? (i = 0, s = 1,
10             printf("\nTabloul sortat:\n")) : (w == 0 ? 1 : (printf("\n"), 0))));
11         (r == 0) && (printf("v[%d] = ", i), scanf("%d", &v[i])),
12         (n > 1) && (r == 1) && (s == 0) && (v[i] > v[j]) && (v[i] = v[i] + v[j] - (v[j] = v[i])),
13         (r == 1) && (s == 1) && (w == 0) && (printf("%d ", v[i])),
14         i++, (i == n) && (s == 1) && (w == 0) && (w = 1),
15         (r == 1) && (s == 0) && (i < n-1) && (j = j < n-1 ? --i, j+1 : i+1));
16
17     return 0;
18 }
```

# Exemple FOR

```
exempluFor2.c ●

1 #include <stdio.h>
2
3 int main()
4 {
5     int i, j, n, r, s, w;
6     int v[100];
7
8     for(i = 0, j = 1, r = s = w = 0, printf("Numarul de elemente: "), scanf("%d", &n), printf("\nElementele:\n");
9         i < n ? 1 : (r == 0 ? (i = 0, r = 1) : (s == 0 ? (i = 0, s = 1,
10             printf("\nTabloul sortat:\n")) : (w == 0 ? 1 : (printf("\n"), 0))));
11         (r == 0) && (printf("v[%d] = ", i), scanf("%d", &v[i])),
12         (n > 1) && (r == 1) && (s == 0) && (v[i] > v[j]) && (v[i] = v[i] + v[j] - (v[j] = v[i])),
13         (r == 1) && (s == 1) && (w == 0) && (printf("%d ", v[i])),
14         i++, (i == n) && (s == 1) && (w == 0) && (w = 1),
15         (r == 1) && (s == 0) && (i < n-1) && (j = j < n-1 ? --i, j+1 : i+1));
16
17     return 0;
18 }
```

Bogdan-Alexes-MacBook-Pro:curs5 bogdan\$ ./a.out

```
Numarul de elemente: 5

Elementele:
v[0] = 10
v[1] = 11
v[2] = 13
v[3] = 14
v[4] = 12

Tabloul sortat:
10 11 12 13 14
```

# Instrucțiunile break, continue și goto

- ❑ realizează **salturi**
  - ❑ îintrerup controlul 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 aceleasi 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)

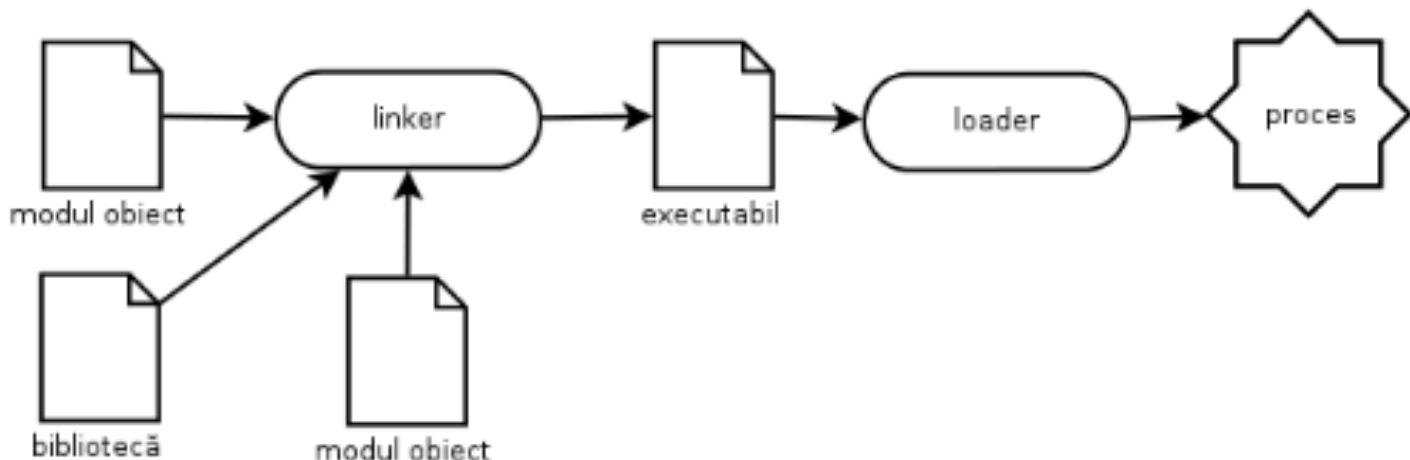
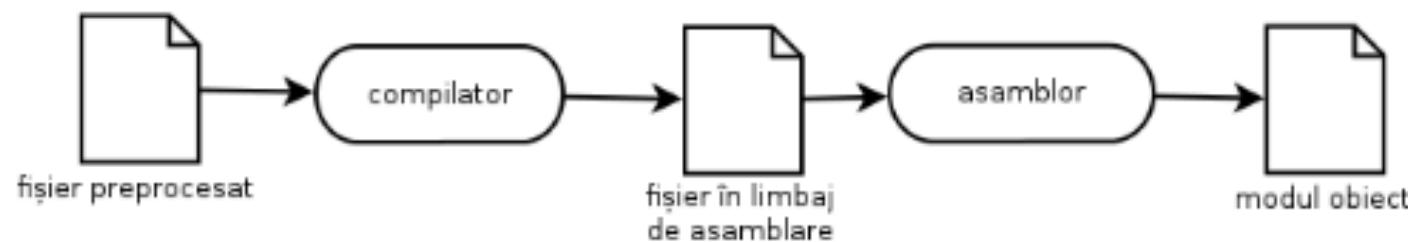
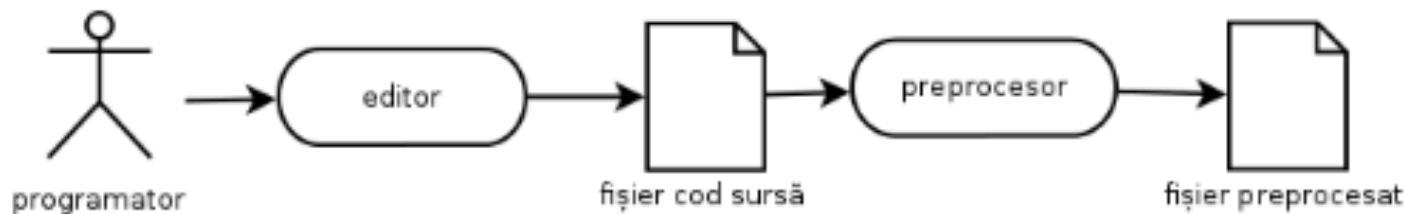
- ❑ are două forme:
  - ❑ return;
  - ❑ return expresie;

```
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
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;
```

# Cuprinsul cursului de azi

1. Instrucțiuni de control (break, continue, goto, return).
2. Etapele realizării unui program C.
3. Directive de preprocessare. Macrodefiniții.
4. Funcții de citire/scriere.

# Etapele realizării unui program în C



# Etapele realizării unui program în C

- ❑ se parcurg următoarele etape pentru obținerea unui cod executabil:
  - ❑ **editarea** codului sursă
    - ❑ salvarea fișierului cu extensia .c
  - ❑ **preprocesarea**
    - ❑ efectuarea directivelor de preprocesare (**#include**, **#define**)
    - ❑ ca un editor – modifică și adaugă la codul sursă
  - ❑ **compilarea**
    - ❑ verificarea sintaxei
    - ❑ codul este tradus din cod de nivel înalt în limbaj de asamblare
  - ❑ **asamblarea**
    - ❑ transformare în cod obiect (limbaj mașină) cu extensia .o, .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

# Etapele realizării unui program în C

## □ Exemplul 1

```
exemplu1.c

1 // program "exemplu1.c" scris de Bogdan Alexe
2 // ultima versiune 30.10.2018
3
4 #include <stdio.h>
5 #include "algebra.c"
6 #define MIN 0
7
8 int main()
9 {
10     int a, b;
11     do
12     {
13         printf("a=");
14         scanf("%d",&a);
15         printf("b=");
16         scanf("%d",&b);
17     } while (a<=MIN || b<=MIN);
18
19     printf("cmmdc(%d,%d) = %d \n", a,b,cmmdc(a,b));
20     return 0;
21 }
```

```
algebra.c

1 int cmmdc(int a, int b)
2 {
3     int c = a%b;
4     while (c)
5     {
6         a = b;
7         b = c;
8         c = a%b;
9     }
10 }
```

DEMO

# Etapele realizării unui program în C

- din linia de comandă (pe Mac):

- **preprocesarea**

- **gcc –E exemplu1.c**

```
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
    const char * restrict, va_list);
# 499 "/usr/include/stdio.h" 2 3 4
# 5 "exemplu1.c" 2
# 1 "./algebra.c" 1
int cmmdc(int a, int b)
{
    int c = a%b;
    while (c)
        {a = b;
         b = c;
         c = a%b;
        }
    return b;
}
# 6 "exemplu1.c" 2

int main()
{
    int a, b;
    do
    {
        printf("a=");
        scanf("%d",&a);
        printf("b=");
        scanf("%d",&b);
    } while (a<=0 || b<=0);

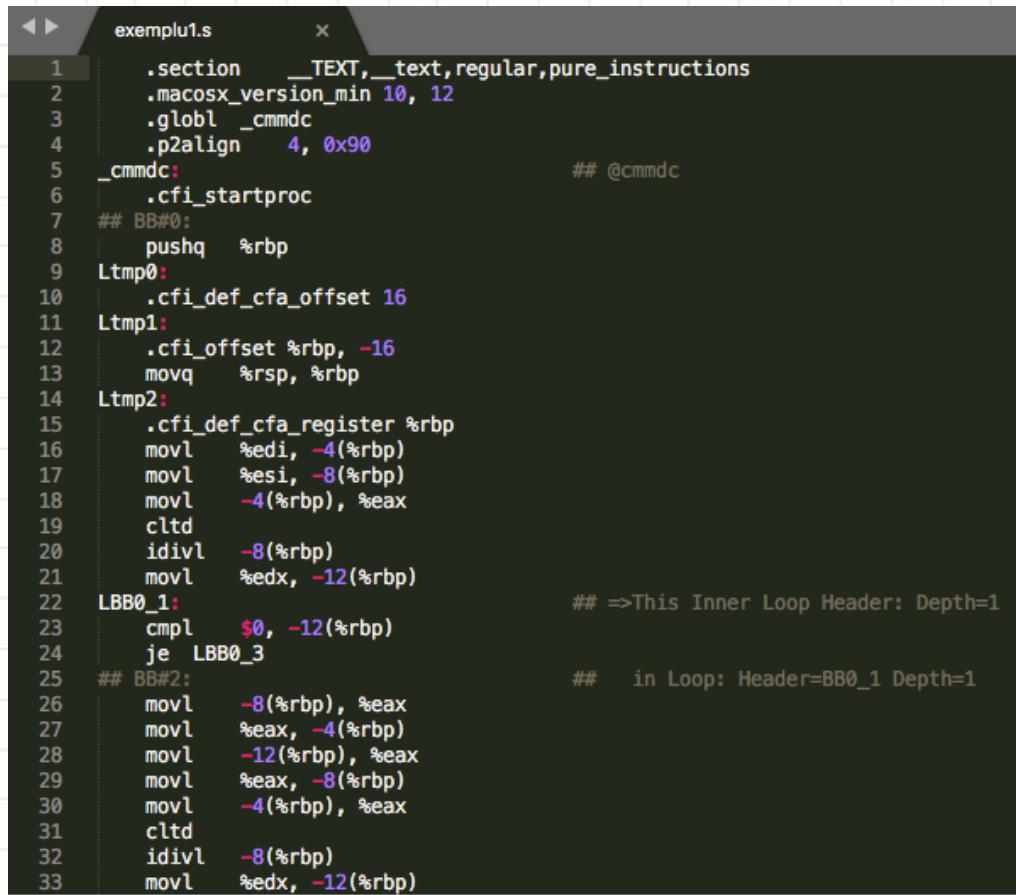
    printf("cmmdc(%d,%d) = %d \n", a,b,cmmdc(a,b));
    return 0;
}
```

# Etapele realizării unui program în C

- din linia de comandă (pe Mac):

- **compilarea**

- gcc –S exemplu1.c (produce exemplu1.s)



```
exemplu1.s      x
1 .section    __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 12
3 .globl _cmmdc
4 .p2align 4, 0x90
5 _cmmdc:          ## @cmmdc
6     .cfi_startproc
7 ## BB#0:
8     pushq %rbp
9 Ltmp0:
10    .cfi_def_cfa_offset 16
11 Ltmp1:
12    .cfi_offset %rbp, -16
13    movq %rsp, %rbp
14 Ltmp2:
15    .cfi_def_cfa_register %rbp
16    movl %edi, -4(%rbp)
17    movl %esi, -8(%rbp)
18    movl -4(%rbp), %eax
19    cltd
20    idivl -8(%rbp)
21    movl %edx, -12(%rbp)
22 LBB0_1:          ## =>This Inner Loop Header: Depth=1
23    cmpl $0, -12(%rbp)
24    je LBB0_3
25 ## BB#2:          ## in Loop: Header=BB0_1 Depth=1
26    movl -8(%rbp), %eax
27    movl %eax, -4(%rbp)
28    movl -12(%rbp), %eax
29    movl %eax, -8(%rbp)
30    movl -4(%rbp), %eax
31    cltd
32    idivl -8(%rbp)
33    movl %edx, -12(%rbp)
```

# Etapele realizării unui program în C

- din linia de comandă (pe Mac):

- **compilarea**

- gcc –S exemplu1.c (produce exemplu1.s)

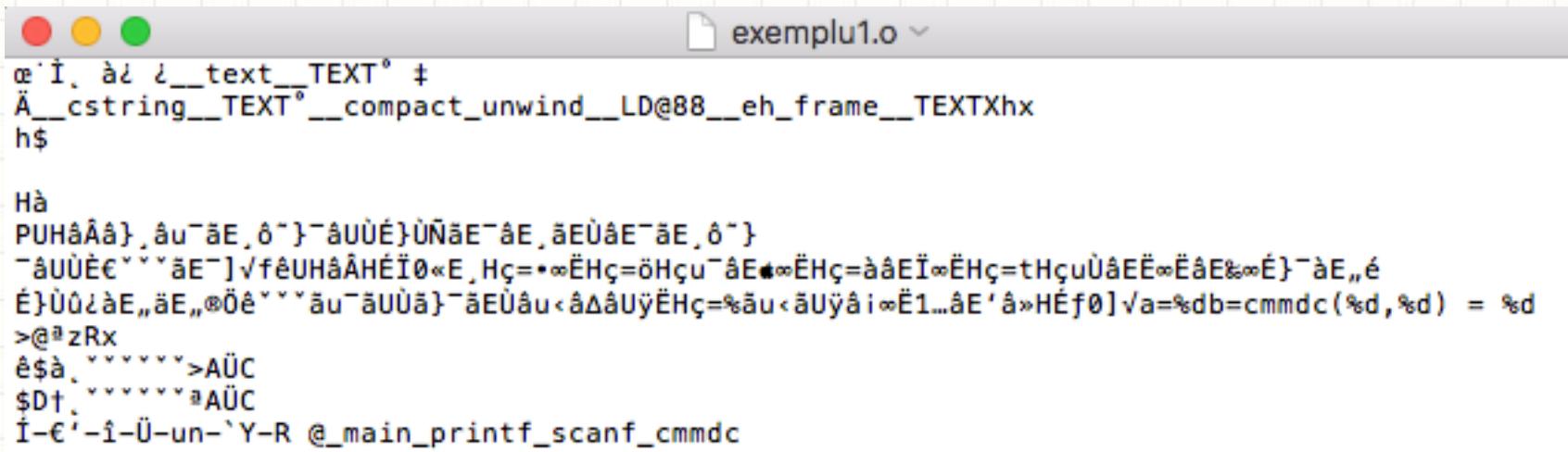
```
35 LBB0_3:  
36     movl    -8(%rbp), %eax  
37     popq    %rbp  
38     retq  
39     .cfi_endproc  
40  
41     .globl  _main  
42     .p2align 4, 0x90  
43 _main:                      ## @main  
44     .cfi_startproc  
45     ## BB#0:  
46     pushq   %rbp  
47 Ltmp3:  
48     .cfi_def_cfa_offset 16  
49 Ltmp4:  
50     .cfi_offset %rbp, -16  
51     movq    %rsp, %rbp  
52 Ltmp5:  
53     .cfi_def_cfa_register %rbp  
54     subq    $48, %rsp  
55     movl    $0, -4(%rbp)  
56 LBB1_1:                      ## =>This Inner Loop Header: Depth=1  
57     leaq    L_.str(%rip), %rdi  
58     movb    $0, %al  
59     callq   _printf  
60     leaq    L_.str.1(%rip), %rdi  
61     leaq    -8(%rbp), %rsi  
62     movl    %eax, -16(%rbp)      ## 4-byte Spill  
63     movb    $0, %al  
64     callq   _scanf  
65     leaq    L_.str.2(%rip), %rdi
```

# Etapele realizării unui program în C

- din linia de comandă (pe Mac):

- **asamblarea**

- gcc –c exemplu1.c (produce exemplu1.o)



```
exemplu1.o
```

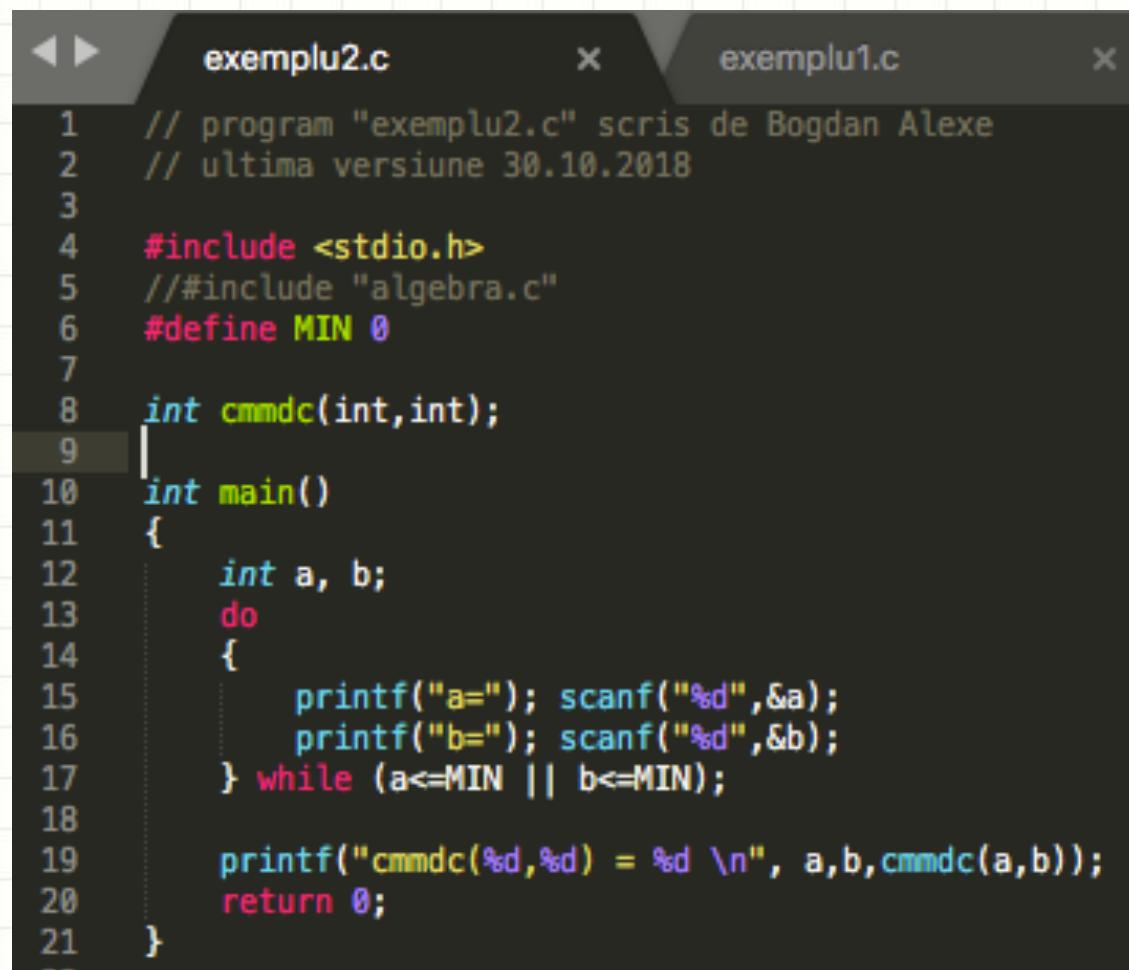
```
.file .text _TEXT
.cstring _TEXT __compact_unwind__LD@88 __eh_frame__TEXTXhx
h$  
  
Hà  
PUHâÀâ}, àu-äE, ô"}-âUÙÉ}ÙÑäE-âE, äEÙâE-äE, ô"}  
-âUÙÉC***äE"]\fêUHâÀHÉÏ0«E, Hç=+∞ËHç=öHç-âE*∞ËHç=àâEÏ∞ËHç=tHçUÙâEË∞ËâE‰∞É}-àE,,é  
É}ÙÙ{àE,,äE,,øÖe***äu-äUÙä}-äEÙâu<âΔâUÿËHç=%äu<äUÿâ i∞Ë1...âE'â»HÉf0]\f=a=%db=cmmdc(%d,%d) = %d  
>@zRx  
é$à,*****>AÜC  
$D† *****@AÜC  
í-€:-i-Ü-un-`Y-R @_main_printf_scandc_cmmdc
```

# Etapele realizării unui program în C

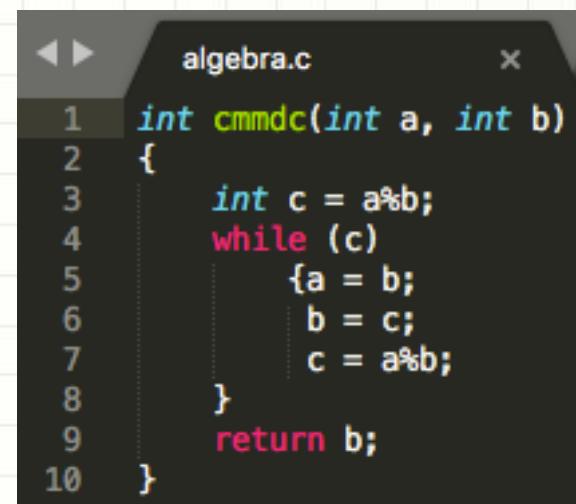
- din linia de comandă (pe Mac):
  - **preprocesare + compilare + asamblare + link-editare**
    - gcc exemplu1.c
      - produce a.out ca fisier executabil
    - gcc exemplu1.c –o exemplu1Executabil
      - produce exemplu1Executabil ca fisier executabil
- În exemplul 1 se copiază funcția cmmdc în fisierul sursă exemplu1.c și apoi se compilează întreg fisierul sursă obținut
- pentru proiecte mari (zeci de mii de linii de cod) dacă schimbăm o funcție nu vrem să recompilăm întreg proiectul ci doar să compilăm fisierul cu funcția schimbată. Link-editorul va produce codul obiect final.
- principiul de programare modulară

# Etapele realizării unui program în C

## □ Exemplul 2



```
00 exemplu2.c x exemplu1.c x
01
02 // program "exemplu2.c" scris de Bogdan Alexe
03 // ultima versiune 30.10.2018
04
05 #include <stdio.h>
06 //#include "algebra.c"
07 #define MIN 0
08
09 int cmmdc(int,int);
10
11 int main()
12 {
13     int a, b;
14     do
15     {
16         printf("a="); scanf("%d",&a);
17         printf("b="); scanf("%d",&b);
18     } while (a<=MIN || b<=MIN);
19
20     printf("cmmdc(%d,%d) = %d \n", a,b,cmmdc(a,b));
21     return 0;
22 }
```



```
00 algebra.c x
01
02 int cmmdc(int a, int b)
03 {
04     int c = a%b;
05     while (c)
06     {
07         a = b;
08         b = c;
09         c = a%b;
10     }
11     return b;
12 }
```

DEMO

# Etapele realizării unui program în C

- compilez fiecare modul în parte (“exemplu2.c”, “algebra.c”)
  - gcc –c algebra.c
    - produce algebra.o
  - gcc –c exemplu2.c
    - produce exemplu2.o
- Link-editez codul obiect (am nevoie ca “exemplu2.c” să știe unde găsește codul de executat pentru funcția “cmmdc”)
  - gcc algebra.o exemplu2.o
    - produce fisierul executabil a.out
  - gcc algebra.o exemplu2.o –o exemplu2Executabil
    - produce fisierul executabil exemplu2Executabil

# Cuprinsul cursului de azi

1. Instrucțiuni de control (break, continue, goto, return).
2. Etapele realizării unui program C.
3. Directive de preprocessare. Macrodefiniții.
4. Funcții de citire/scriere.

# Preprocesare în limbajul C

- ❑ preprocesarea apare înaintea procesului de compilare a codului sursă (fișier text editat într-un editor și salvat cu extensia .c).
- ❑ preprocesarea codului sursă asigură:
  - ❑ includerea conținutului fișierelor (de obicei a fișierelor *header*)
  - ❑ definirea de macro-uri (macrodefiniții)
  - ❑ compilarea condiționată
- ❑ constă în substituirea simbolurilor din codul sursă pe baza directivelor de preprocesare
- ❑ directivele de preprocesare sunt precedate de caracterul diez #

# Directiva #include

- copiază conținutul fișierului specificat în textul sursă
- `#include <nume_fisier>`
  - caută nume\_fisier în directorul unde se află fișierele din librăria standard instalată odată cu compilatorul
- `#include “nume_fisier”`
  - caută nume\_fisier în directorul curent

# Directiva #define

```
#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);
    int m=MIN(x,y);
    printf("%d\n",m);
    int a=ABS1(x-y);
    int b=ABS2(x-y);
    printf("%d %d\n",a,b);
    INTER(int,a,b);
    printf("%d %d\n",a,b);
    INTER(int,a,b);
    printf("%d %d\n",a,b);
    return 0;
}
```

# Directiva #define

```
#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;
}
```

# Directiva #define

- ❑ folosită pentru definirea (înlocuirea) constantelor simbolice și a macro-urilor
- ❑ definirea unei **constante simbolice** este un caz special al definirii unui macro (fragment de cod care primește un nume)  
`#define nume text`
- ❑ în timpul preprocesării **nume** este înlocuit cu **text**
- ❑ **text** poate să fie mai lung 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

# Directiva `#define`

- ❑ folosită pentru definirea (înlocuirea) constantelor simbolice și a macro-urilor
- ❑ definirea unei **constante simbolice** este un caz special al definirii unui macro (fragment de cod care primește un nume)

```
#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`

# Directiva #define

- ❑ 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)**

# Directiva #define

- ❑ 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

```
1 #include <stdio.h>
2
3 #define VERSION 2
4
5 int main()
6 {
7
8     #if VERSION == 1
9     {
10         printf ("versiunea 1 \n");
11         printf ("Adaugam modulele pentru versiunea 1 ... \n");
12         // continua cu includerea diverselor module pentru versiunea 1
13     }
14
15     #elif VERSION == 2
16     {
17         printf ("versiunea 2 \n");
18         printf ("Adaugam modulele pentru versiunea 2 ... \n");
19         // continua cu includerea diverselor module pentru versiunea 2
20     }
21     #elif VERSION == 3
22     {
23         printf ("versiunea 3 \n");
24         printf ("Adaugam modulele pentru versiunea 3 ... \n");
25         // continua cu includerea diverselor module pentru versiunea 3
26     }
27
28     #endif
29     return 0;
30
31 }
```

# 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 (#elif)  
    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

## STDIO.H

```
57  *
58  *  @(#)stdio.h 8.5 (Berkeley) 4/29/95
59  */
60
61 #ifndef _STDIO_H_
62 #define _STDIO_H_
63
```

```
496 #if defined (__GNUC__) && _FORTIFY_SOURCE > 0 && !defined (__cplus)
497 /* Security checking functions.  */
498 #include <secure/_stdio.h>
499 #endif
500
501 #endif /* _STDIO_H_ */
502
```

# 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`:

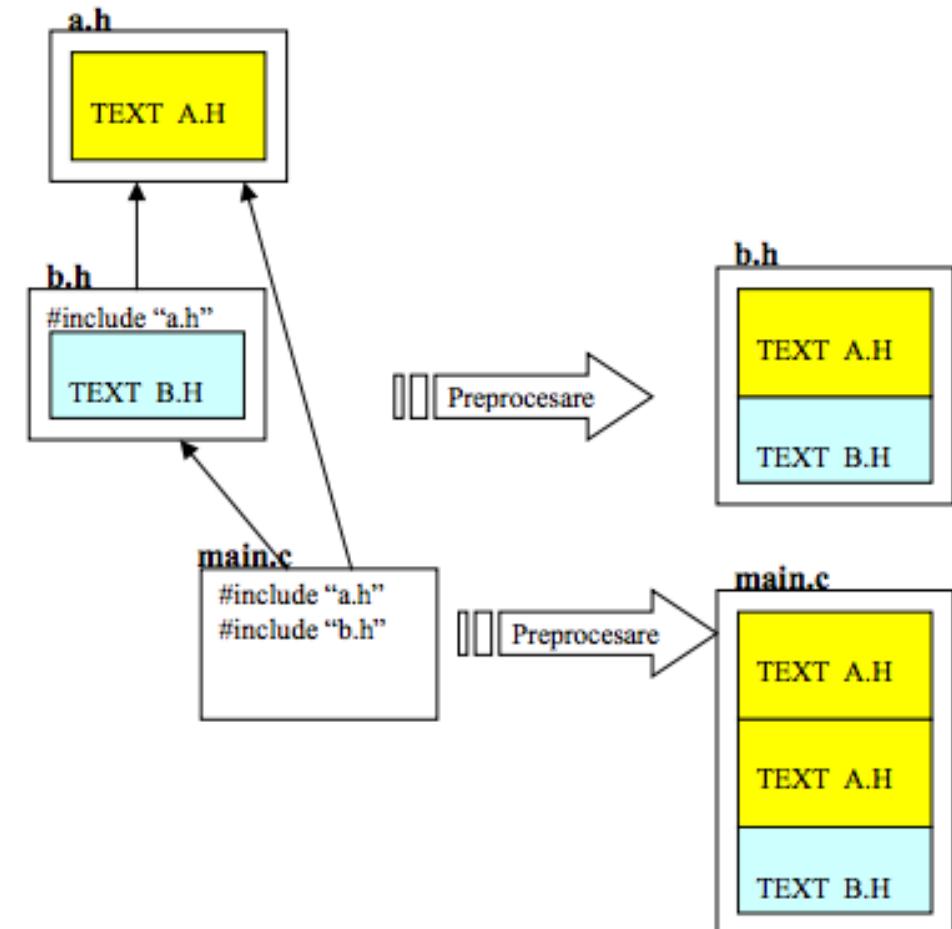
```
#ifndef nume  
    text  
#endif
```

```
#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
- fișier antet “a.h”
- fișier antet “b.h”
  - include pe “a.h”
- main include “a.h” și “b.h”



# Compilarea condiționată

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

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

# Macro-uri predefinite

- există o serie de macro-uri predefinite care nu trebuie re/definite:

<u>DATE</u>	data compilării
<u>CDECL</u>	apelul funcției urmărește convențiile C
<u>STDC</u>	definit dacă trebuie respectate strict regulile ANSI C
<u>FILE</u>	numele complet al fișierului curent compilat
<u>FUNCTION</u>	numele funcției curente
<u>LINE</u>	numărul liniei curente

```
#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
#ifndef 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;
}
```

# Cuprinsul cursului de azi

1. Instrucțiuni de control (break, continue, goto, return).
2. Etapele realizării unui program C.
3. Directive de preprocessare. Macrodefiniții.
4. Funcții de citire/scriere.

# Functii de citire și scriere

- operații de **citire** și **scriere** în C:
  - de la **tastatură** (stdin) și la **ecran** (stdout);
  - **prin fișiere**;
  - efectuate cu ajutorul funcțiilor de bibliotecă
- funcții pentru **citirea de la tastatură** și **scrierea la ecran**
  - fără formatare: **getchar**, **putchar**, **getch**, **getche**, **putch**, **gets**, **puts**
  - cu formatare: **scanf**, **printf**
  - incluse în bibliotecile **stdio.h** (**getchar**, **putchar**, **gets**, **puts**, **scanf**, **printf**) sau **conio.h** (**getch**, **getche**, **putch**)
  - CODE::BLOCKS nu include biblioteca **conio.h**

# Functiile getchar și putchar

- operatii de **citire** și **scriere** a caracterelor:
  - **int getchar(void)** - citește un caracter de la tastatură. Așteaptă până este apasată o tastă și returnează valoarea sa → tasta apăsată are imediat ecou pe ecran.
  - **int putchar(int c)** - scrie un caracter pe ecran în poziția curentă a cursorului
  - fișierul antet pentru aceste funcții este **stdio.h.**

# Functiile getchar și putchar

## ❑ exemplu:

```
exempluCitireScriere1.c  x
01
02
03
04
05
06
07
08
09
10
11
12
13
14
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int optiune;
7     printf("Alegeti DA sau NU. Optiunea este : ");
8     optiune = getchar();
9     putchar(optiune);
10
11     printf("\n");
12
13     return 0;
14 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
Alegeti DA sau NU. Optiunea este : DA
| D
```

# Functiile gets și puts

- operații de **citire și scriere** a sirurilor de caractere:
  - **char \*gets(char \*s)** – citește caractere din **stdin** și le depune în zona de date de la adresa s, până la apăsarea tastei Enter. În sir, tastei Enter îi va corespunde caracterul '\0'.
    - dacă operația de citire reușește, funcția întoarce adresa sirului, altfel valoarea **NULL** (= 0).
  - **int puts(const char \*s)** – scrie pe ecran sirul de la adresa s sau o constantă sir de caractere și apoi trece la linie nouă.
    - dacă operația de scriere reușește, funcția întoarce ultimul caracter, altfel valoarea **EOF** (-1).
  - fișierul antet pentru aceste funcții este **stdio.h**

# De ce să nu folosiți funcția gets

- **char \*gets(char \*s)**
- primește ca input numai un buffer (**s**), nu stim dimensiunea lui
- problema de buffer overflow: citim în **s** mai mult decât dimensiunea lui, **gets** nu ne impiedică, scrie datele în alta parte
- folositi fgets: **char \*fgets(char \*s, int size, FILE \*stream)**
  - **fgets(buffer, sizeof(buffer), stdin);**
- în standardul C11 funcția gets este eliminată

# Functiile printf și scanf

- funcții de citire (**scanf**) și scriere (**printf**) cu formatare;
- formatarea specifică conversia datelor de la reprezentarea externă în reprezentarea internă (**scanf**) și invers (**printf**);
- formatarea se realizează pe baza descriptorilor de format
  - **%[flags][width][.precision][length]specifier**
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>

# Functia printf

## □ prototipul funcției:

*int printf( const char \*format, argument1, argument2, ...);*

unde:

- **format** este un sir de caractere ce definește textele și formatele datelor care se scriu pe ecran
- **argument1, argument2,...** sunt expresii. Valorile lor se scriu pe ecran conform specificatorilor de format prezenți în format
- functia **printf** întoarce numărul de octeți transferați sau EOF (-1) în caz de eșec.

# Modelatori de format

- mulți specicatori de format pot accepta modelatori care modifică ușor semnificația lor:
  - alinierea la stânga
  - minim de mărime a câmpului
  - numărul de cifre zecimale
  
- modelatorul de format se află între semnul procent și codul pentru format:
  - caracterul ‘–’ specifică aliniere la stânga;
  - sir de cifre zecimale specifică dimensiunea câmpului pentru afișare
  - caracterul ‘.’ urmat de cifre specifică precizia reprezentării

# Modelatorul specifier

- formatarea se realizează pe baza descriptorilor de format
  - `%[flags][width][.precision][length]specifier`
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>
  - **specifier**

Specifier de format	Reprezentare
<code>%c</code>	caracter
<code>%s</code>	șir de caractere
<code>%d, %i</code>	întreg în zecimal
<code>%u</code>	întreg în zecimal fără semn
<code>%o</code>	întreg în octal
<code>%x</code>	întreg în hexazecimal fără semn (litere mici)
<code>%X</code>	întreg în hexazecimal fără semn (litere mari)
<code>%f</code>	număr real în virgulă mobilă
<code>%e, %E</code>	notație științifică – o cifră la parte întreagă
<code>%ld, %li, %lu, %lo, %lx</code>	cu semnificațiile de mai sus, pentru întregi lungi
<code>%p</code>	pointer

# Modelatorul flags

- formatarea se realizează pe baza descriptorilor de format
  - `%[flags][width][.precision][length]specifier`
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>
  - flags

flags	description
-	Left-justify within the given field width; Right justification is the default (see <i>width</i> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see <i>width</i> sub-specifier).

# Modelatorul width

- formatarea se realizează pe baza descriptorilor de format
  - `%[flags][width][.precision][length]specifier`
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>
  - **width**

<b>width</b>	<b>description</b>
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The <i>width</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

# Modelatorul precision

- formatarea se realizează pe baza descriptorilor de format
  - `%[flags][width][.precision][length]specifier`
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>
  - precision

.precision	description
.number	<p>For integer specifiers (d, i, o, u, x, x): <i>precision</i> specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A <i>precision</i> of 0 means that no character is written for the value 0.</p> <p>For a, A, e, E, f and F specifiers: this is the number of digits to be printed <b>after</b> the decimal point (by default, this is 6).</p> <p>For g and G specifiers: This is the maximum number of significant digits to be printed.</p> <p>For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered.</p> <p>If the period is specified without an explicit value for <i>precision</i>, 0 is assumed.</p>
.*	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

# Modelatorul length

- formatarea se realizează pe baza descriptorilor de format
  - %[flags][width][.precision][length]specifier
  - detalii aici: <http://www.cplusplus.com/reference/cstdio/printf/>
  - length

length	specifiers							
	d i	u o x X	f F e E g G a A	c	s	p	n	
(none)	int	unsigned int	double	int	char*	void*	int*	
hh	signed char	unsigned char						signed char*
h	short int	unsigned short int						short int*
l	long int	unsigned long int		wint_t	wchar_t*			long int*
ll	long long int	unsigned long long int						long long int*
j	intmax_t	uintmax_t						intmax_t*
z	size_t	size_t						size_t*
t	ptrdiff_t	ptrdiff_t						ptrdiff_t*
L			long double					

# Modelatori de format pentru printf

## exemplu 1:



The screenshot shows a code editor window with a dark theme. The title bar says "exempluPrintf1.c". The code itself is as follows:

```
#include <stdio.h>
int main()
{
    double numar;
    numar = 10.1234;
    printf("numar = %f\n", numar);
    printf("numar = %10f\n", numar);
    printf("numar = %012f\n", numar);

    printf("%.4f\n", 123.1234567);
    printf("%8.3d\n", 1000);
    printf("%3.8d\n", 1000);
    printf("%+10d\n", 1000);
    printf("%-+10d\n", 1000);
    printf("%10.15s\n", "Acesta este un test simplu");

    return 0;
}
```

# Modelatori de format pentru printf

## exemplu 1:

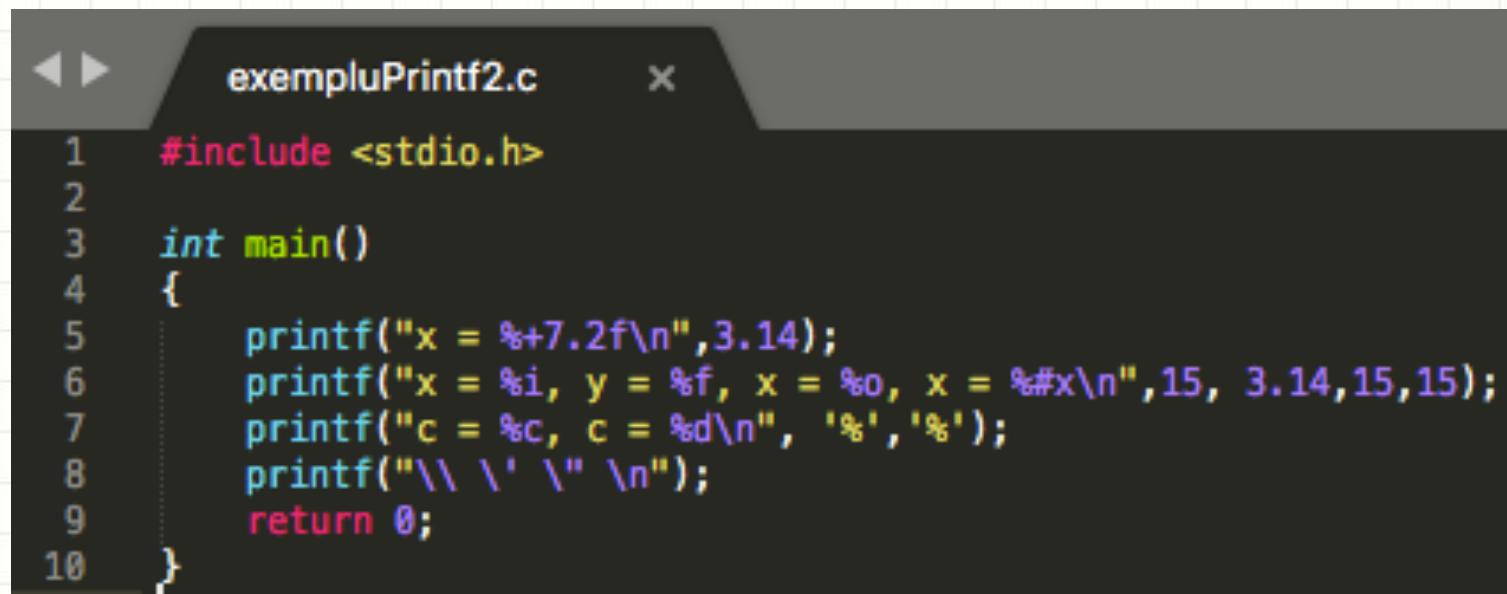
```
exempluPrintf1.c
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     double numar;
6     numar = 10.1234;
7
8     printf("numar = %f\n", numar);
9     printf("numar = %10f\n", numar);
10    printf("numar = %012f\n", numar);
11
12    printf("%.4f\n", 123.1234567);
13    printf("%8.3d\n", 1000);
14    printf("%3.8d\n", 1000);
15    printf("%+10d\n", 1000);
16    printf("%-+10d\n", 1000);
17    printf("%10.15s\n", "Acesta este un");
18
19    return 0;
20 }
```

```
Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ gcc exempluPrintf1.c
Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
numar = 10.123400
numar = 10.123400
numar = 00010.123400
123.1235
      1000
      00001000
          +1000
      +1000
Acesta este un
```

# Modelatori de format pentru printf

## □ exemplu 2:

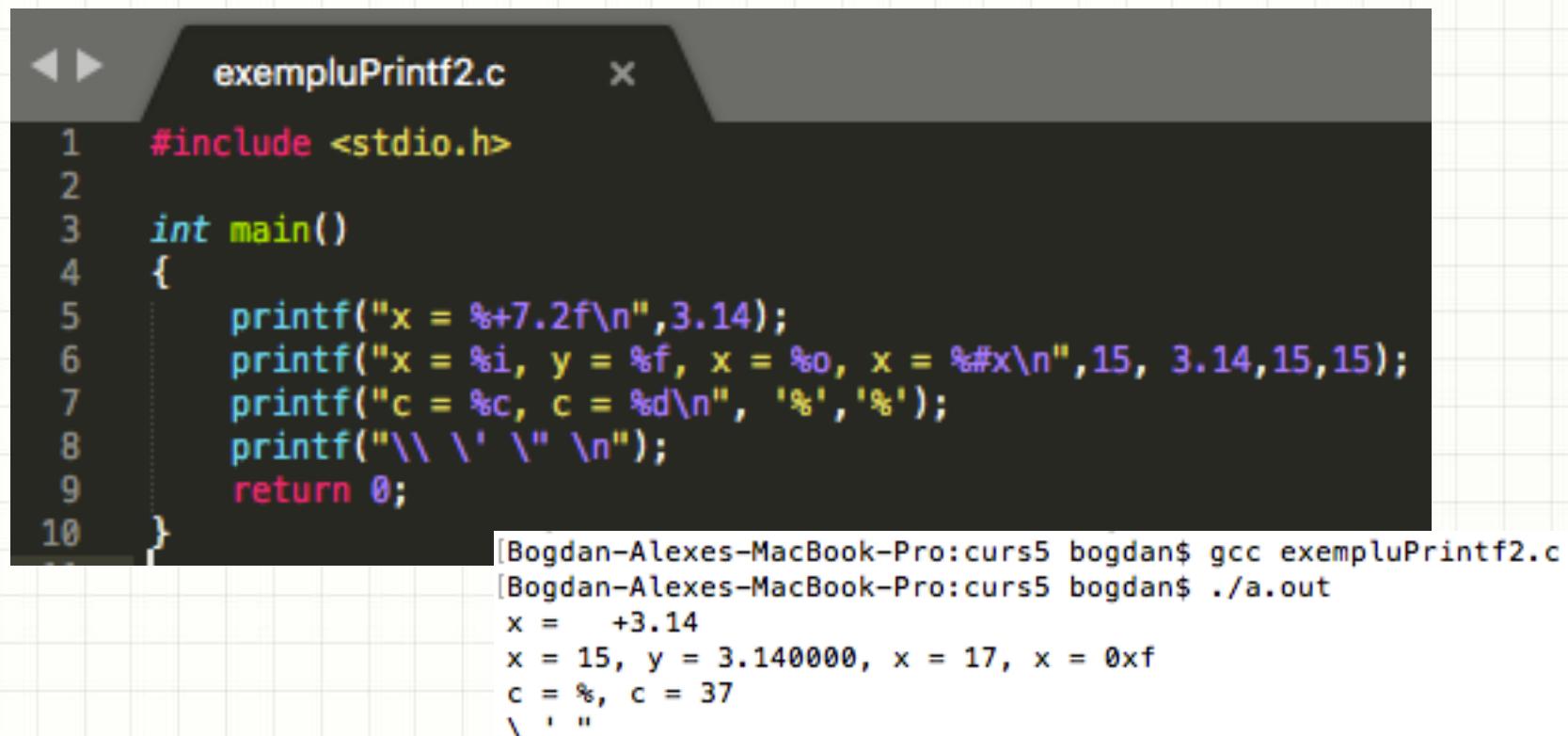


```
exempluPrintf2.c      x

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("x = %+7.2f\n", 3.14);
6     printf("x = %i, y = %f, x = %o, x = %#x\n", 15, 3.14, 15, 15);
7     printf("c = %c, c = %d\n", '%', '%');
8     printf("\\ \\ \\ \\ \n");
9     return 0;
10 }
```

# Modelatori de format pentru printf

## □ exemplu 2:



```
exempluPrintf2.c      x

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("x = %+7.2f\n",3.14);
6     printf("x = %i, y = %f, x = %o, x = %#x\n",15, 3.14,15,15);
7     printf("c = %c, c = %d\n", '%','%');
8     printf("\\ \\ \" \" \n");
9     return 0;
10 }
```

[Bogdan-Alexes-MacBook-Pro:curs5 bogdan\$ gcc exempluPrintf2.c  
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan\$ ./a.out  
x = +3.14  
x = 15, y = 3.140000, x = 17, x = 0xf  
c = %, c = 37  
\ \"

# Functia scanf

- **prototipul functiei:**

***int scanf( const char \* format ,adresa1, adresa2, ...);***

unde:

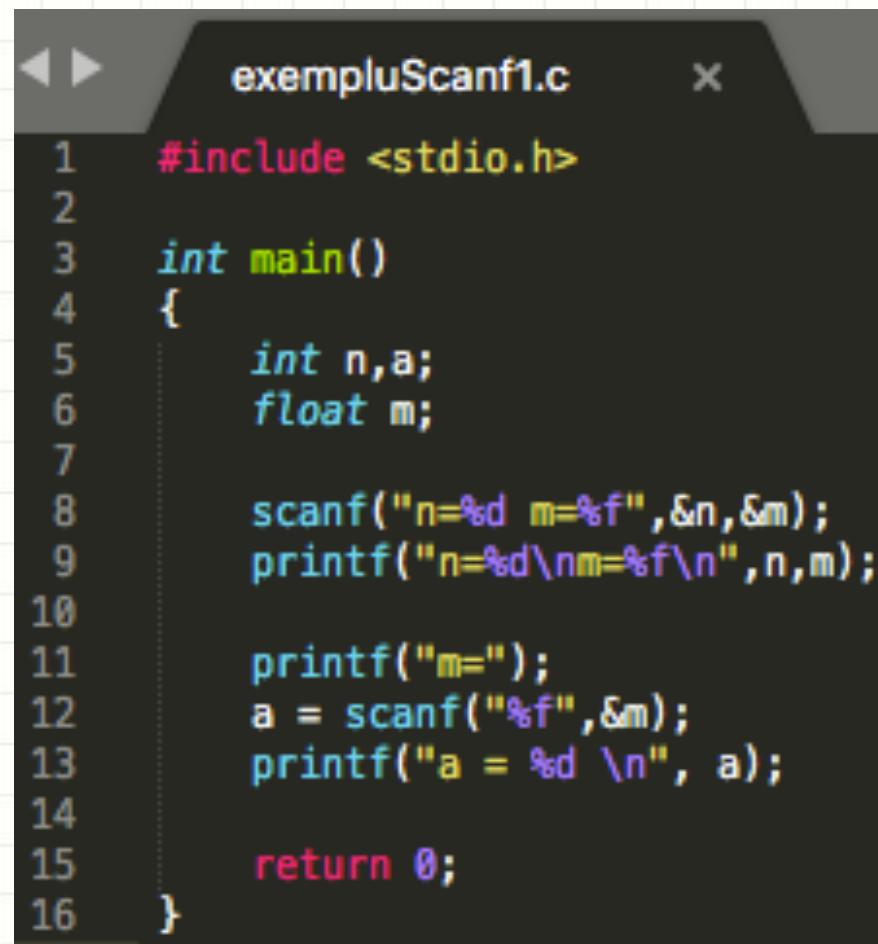
- **format** este un sir de caractere ce defineste textele si formatele datelor care se citesc de la tastatura
- **adresa1, adresa2,...** sunt adresele zonelor din memorie in care se pастreaza datele citite după ce au fost convertite din reprezentarea lor externă în reprezentare internă.
- functia **scanf** întoarce numărul de câmpuri citite și depuse la adresele din listă. Dacă nu s-a stocat nici o valoare, funcția întoarce 0.

# Functia scanf

- sirul de formatare (format) poate include următoarele elemente:
  - spațiu alb: funcția citește și ignoră spațiile albe (spațiu, tab, linie nouă) înaintea următorului caracter diferit de spațiu
  - un singur spațiu în sirul de formatare se suprapune asupra oricărora spații din sirul introdus, inclusiv asupra nici unui spațiu
  - caracter diferit de spațiu, cu excepția caracterului %: funcția citește următorul caracter de la intrare și îl compară cu caracterul specificat în sirul de formatare
    - dacă se potrivește, funcția are succes și trece mai departe la citirea următorului caracter din intrare
    - dacă nu se potrivește, funcția eșuează și lasă următoarele caractere din intrare nepreluate

# Functia scanf

## □ exemplu 1:



```
exempluScanf1.c
```

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n,a;
6     float m;
7
8     scanf("n=%d m=%f",&n,&m);
9     printf("n=%d\nm=%f\n",n,m);
10
11    printf("m=");
12    a = scanf("%f",&m);
13    printf("a = %d \n", a);
14
15    return 0;
16 }
```

# Functia scanf

## □ exemplu 1:

```
exempluScanf1.c      x
1 #include <stdio.h>
2
3 int main()
4 {
5     int n,a;
6     float m;
7
8     scanf("n=%d m=%f",&n,&m);
9     printf("n=%d\nm=%f\n",n,m);
10
11    printf("m=");
12    a = scanf("%f",&m);
13    printf("a = %d \n", a);
14
15    return 0;
16 }
```

```
Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
n=25      m=2.6
n=25
m=2.600000
m=i37
a = 0
```