

Capitolul 11

DIRECTIVE PREPROCESOR.

FUNCȚII CU NUMĂR VARIABIL DE PARAMETRI

- Directiva *#define*
- Directiva *#include*
- Directivele *#if*, *#elif*, *#else* si *#endif*
- Directiva *#undef*
- Macro-instrucțiuni predefinite
- Funcții cu număr variabil de parametri

Directivele către preprocesor constituie un fel de limbaj inclus în limbajul C. În esență, se deosebesc de instrucțiunile limbajului prin următoarea particularitate:

- **Instrucțiunea** reprezintă o comandă pentru **procesor**
- **Directiva către preprocesor** reprezintă o comandă pentru **compiler** (**preprocesor**), prin care i se cere acestuia să execute ceva **înainte** de a începe compilarea propriu-zisă

11.1 Directiva #define

Directiva **#define** este folosită, în general, pentru:

- Definire de **constante**
- Scriere de **macro-definiții (macro-instrucțiuni)**

11.1.1 Definire de **constante**

Numele (identificatorul) ce se asociază constantei. Prin convenție, se scrie cu litere mari, pentru a se deosebi de numele variabilelor, funcțiilor etc.

#define **PI** 3.14159

← **Nu** se scrie punct și virgulă!

↑
spațiu

text

Compilarea programelor sursă scrise cu ajutorului limbajului C se desfășoară astfel:

- compilatorul caută mai întâi directivele preprocesor, pe care le identifică datorită simbolului **#**, și efectuează comenzile cerute de acestea
- în cazul directivei **#define** din exemplul anterior, **înlocuiește** în programul sursă **toate** aparițiile identificatorului **PI** cu textul **3.14159**
- compilează programul sursă astfel modificat

Principalele **avantaje** ale utilizării directivei **#define** pentru definirea constantelor:

- claritatea programului sursă
- programul executabil este mai rapid și mai compact decât în cazul utilizării calificatorului **const** la declararea variabilelor;
- ușurința modificării valorii constantei, pentru o eventuală re-compilare a programului sursă

Example:

```
#define PI 3.14159
...
float r, lungime, aria;
...
lungime = 2.0 * PI * r;
aria=PI * r * r;
```

sau

```
#define AND &&
#define OR ||
...
int x;
...
if (x>0 AND x<10) ...
```

sau

```
#define PI 3.141592654
#define DOI_PI 2.0*PI
```



Se pot scrie și în ordine
inversă! (vezi obs. următoare)

Observații:

- Numele definite cu ajutorul directivei **#define** se comportă altfel decât variabilele. Ele **pot fi folosite oriunde în program**, începând cu punctul în care au fost definite, indiferent că acesta este în interiorul sau în afara unei funcții. Important este doar ca expresia rezultată prin înlocuirea numelui cu textul asociat să fie corectă, în contextul programului C respectiv.
- În definirea unei entități pot fi folosite alte definiții, indiferent de ordinea în care au fost scrise, cu condiția ca toate numele (identificatorii) să fie cunoscute **în momentul utilizării** în program a entității respective.
- **Directivele #define pot fi scrise oriunde în program**, chiar dacă, de obicei, se preferă să fie incluse la începutul acestuia.

Exemplu - pentru testare an bisect:

```
#define E_AN_BISECT an%4==0 && an%100!=0 \
|| an%400==0
```

```
...
int an=2011;
#define ESTE {printf("Anul este bisect\n");}
#define NU_ESTE {printf("Anul nu este bisect\n");}
if (E_AN_BISECT) ESTE else NU_ESTE
an=2000;
if (E_AN_BISECT) ESTE else NU_ESTE
...
```

Se observă că este permisă continuarea definiției pe linii succesive, ca la șirurile de caractere

Prin înlocuire, programul sursă anterior devine:

```
int an=2011;
if (an%4==0 && an%100!=0 \
|| an%400==0) {printf("Anul este bisect\n");} else {printf("Anul nu este bisect\n");}
an=2000;
if (an%4==0 && an%100!=0 \
|| an%400==0) {printf("Anul este bisect\n");} else {printf("Anul nu este bisect\n");}
...
```


11.1.2 Macro-definiții (macro-instrucțiuni)

↓ Obligatoriu, fără spațiu!!!

```
#define E_AN_BISECT(y) y%4==0&& y%100!=0|| y%400==0
```



nu este necesar să se declare tipul argumentului

```
if (E_AN_BISECT(1997)) ESTE else NU_ESTE
```

Avantaj:

```
#define LA_PATRAT(x) x*x
```

```
int y, v=2;
```

```
double z, w=3.2;
```

```
y = LA_PATRAT(v);
```

/*va pune în **y** valoarea **v*v**, indiferent de tipul lui v, spre deosebire de cazul unei funcții, la care suntem obligați să precizăm tipul parametrului la definire și să îl respectăm la apelare */

```
z = LA_PATRAT(w);
```

Obs. Care este rezultatul expresiei **y = LA_PATRAT(v+1); ???**

Răspuns: $y = v+1*v+1$!!!

Adică NU $y = (v+1)*(v+1)$, cum intenționam de fapt !!!

Rezultă că, pentru a obține un răspuns corect, ar trebui să rescriem macro-definiția cu **încadrarea fiecărui argument între paranteze**, astfel:

```
#define LA_PATRAT(x) (x)*(x)
```

Totuși, pentru:

```
#define SUM(x,y) (x)+(y)
```

```
...
```

```
rez = 10*SUM(3,4);
```

valoarea calculată va fi $10*3+4$ sau $10*(3+4)$???

Răspuns: **10*3+4 !!!**

Adică NU **10*(3+4)**, cum intenționam de fapt !!!

Pentru a obține un răspuns corect, ar trebui să rescriem macro-definiția cu **încadrarea întregii expresii (a întregului text) între paranteze**, astfel:

```
#define SUM(x,y) ((x)+(y))
```

Ca **regulă generală**, în cazul macro-definițiilor se vor folosi **întotdeauna** paranteze atât pentru încadrarea fiecărui argument în parte, cât și pentru încadrarea întregii expresii (text).

Exemplu:

```
#define MAX(a,b) (((a)>(b))?(a):(b))  
...  
int x, y, val_min, z;  
...  
limita = MAX(x+y, val_min);  
limita = MAX(x,y)*100;  
limita = MAX(x&y,z);  
    /* toate cele trei atriburi se vor executa corect */
```

Observații:

- Macro-definițiile pot avea unul sau mai multe argumente;
- Fiecare argument poate fi utilizat de mai multe ori în cadrul textului macro-definiției;
- Toate argumentele declarate trebuie utilizate în cadrul textului. De exemplu, macro-definiția **#define ADUNA(x, y, z) ((x) + (y))** este incorectă, deoarece nu utilizează în text argumentul **z**;
- Numărul de argumente specificat în momentul folosirii macro-definiției trebuie să corespundă celui de la definire.

În cadrul macro-definițiilor se poate utiliza operatorul # care transformă argumentul macro-definiției în constantă șir de caractere, ca în exemplul următor:

```
#define AFISARE(x) printf(#x)
AFISARE(Buna ziua);
```

Acest fragment de program este echivalent cu:

```
printf("Buna ziua");
```

Exemplu preluat din “Teach Yourself C in 21 Days”:

```
#include <stdio.h>
#define OUT(x) printf(#x " este egal cu %d.\n", x)
void main(void)
{ int valoare = 123;
  OUT(valoare);
}
```

Pe ecran se va afișa

valoare este egal cu 123.

deoarece ultima linie de program va fi înlocuită cu
printf("valoare" " este egal cu %d.\n", valoare);

Un alt operator ce poate fi utilizat în scrierea macro-definițiilor este operatorul de concatenare, ##.

Acesta concatenează două șiruri de caractere în cadrul macro--definiției din care face parte, ca în exemplul următor:

```
#define TEST(x) functie##x  
rezultat = TEST(3)(q, w);
```

Ca efect al pre-procesării, ultima instrucțiune va fi înlocuită cu:

```
rezultat = functie3(q, w);
```

Dacă în programul respectiv sunt definite mai multe funcții cu același prototip, numite, de exemplu, *functie1()*, *functie2()*, *functie3()*, operatorul de concatenare permite apelarea acestora într-un mod flexibil.

➤ **Macro-definiții:**

- Codul lor (textul) se inserează efectiv în program, oriunde și ori-de-câte-ori sunt folosite. Un efect imediat este creșterea dimensiunii programului (programul ocupă mai multă memorie);
- Programul este mai rapid decât în cazul utilizării funcțiilor (execuția programului durează mai puțin)
- Prin folosirea unui număr mic de macro-definiții, programul sursă devine mai ușor de citit. Prin folosirea unui număr mare de macro-definiții, efectul este invers!

➤ **Funcții:**

- Codul sursă corespunzător unei funcții apare o singură dată în program, chiar dacă funcția este apelată de mai multe ori în cadrul acestuia. Efectul imediat este acela că programul ocupă memorie mai puțină decât în cazul utilizării unei macro-definiții
- Programul este mai lent decât în cazul folosirii macro-definițiilor (execuția programului durează mai mult) deoarece se lucrează prin stivă, se salvează contextul funcției apelante etc.

11.2 Directiva #include

... este folosită pentru includerea conținutului unui fișier în cadrul fișierului în care este scrisă, exact în locul în care este scrisă.

De exemplu, dacă dorim să scriem mai multe programe C în care, în mod repetat, trebuie calculate ariile unor figuri geometrice (cerc, dreptunghi, triunghi), putem crea cu ajutorul oricărui editor simplu de text un fișier cu următorul conținut:

```
#define PI 3.14159
#define ARIA_CERC(raza) (PI*(raza)*(raza))
#define ARIA_DREPT(lung, lat) ((lung)*(lat))
#define ARIA_TRI(baza, inalt) ((baza)*(inalt)/2.0)
```

Salvând în memorie fișierul astfel creat și numindu-l, de exemplu, **arii.h** putem crea apoi programele C dorite, având grijă să scriem în ele și directiva **#include "arii.h"** **înainte** de prima utilizare a macro-definițiilor.

Presupunând că numele unuia dintre aceste programe este **matematica.C**, la compilarea sa preprocesorul va căuta fișierul **arii.h în directorul curent și îi va insera conținutul în fișierul sursă matematica.C, exact în locul apariției directivei.**

Căutarea fișierului **arii.h** se face în directorul curent deoarece în cadrul directivei au fost folosite **ghilimele** pentru încadrarea numelui de fișier.

- În general, fişierele folosite pentru includere cu ajutorul directivei specializate conţin definiţii de tip (cu folosirea cuvântului cheie **typedef**), macro-definiţii, definiri de constante, prototipuri de funcţii etc. şi primesc nume cu extensie **.h**, având semnificaţia de **header** (engl.), adică antet, descriere de elemente. Rolul acestor fişiere este deosebit de important în programarea modulară (**detalii în capitolul 12**) dar şi în paradigmele de programare ulterioare acesteia.
- În cadrul unui acelaşi fişier pot fi folosite mai multe directive **#include**, adică pot fi incluse mai multe fişiere distincte.
- Dacă în cadrul directivei se încadrează numele fişierului cu ajutorul parantezelor unghiulare, **< >**, căutarea fişierului ce trebuie inclus se va face în directorul sau directoarele speciale ale sistemului (*Standard Header Directory*) unde sunt memorate fişierele *header* proprii compilatorului ce conţin informaţii necesare pentru folosirea în programe a funcţiilor de bibliotecă.

De exemplu, **#include <stdio.h>** va permite utilizarea funcţiilor din biblioteca descrisă prin **stdio.h**: *printf()*, *scanf()*, *puts()* etc.

Exemplificare pe calculator...

Exemplu didactic - efectul utilizării directivei `#include`

A. Edităm fișierul `test.c` în directorul curent:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
    #include "D:\exemple\test.h"
```

B. Edităm fișierul `test.h` în directorul `D:\exemple`:


```
int i=13;
printf("%d\n", i);
getch();
return 0;
}
```

Exemplificare
pe calculator...

Varianta 1

- Compilare `test.c` => OK
- Link-editare și lansare în execuție `test.exe` => OK

Varianta 2 – pentru Dev C++

- Modificare **test.c**: se scrie **#include "test.h"** în loc de **"D:\exemple\test.h"**
- În meniul **Tools**, sub-meniurile **Compiler Options**, **Directories**, **C Includes**, se caută cu „**Browse for folder**”  și se adaugă numele de director **D:\exemple**
- Compilare **test.c modificat** => OK
- Link-editare și lansare în execuție **test.exe A modificat** => OK

Exemplificare pe calculator...

11.3 Directivele #if, #elif, #else și #endif

- Permit programatorului să impună modul în care să se realizeze compilarea programului sursă. Sunt directive de **compilare condiționată** ce permit **compilarea** unor blocuri de program C numai dacă anumite condiții sunt îndeplinite.
- Efectul poate fi înțeles prin analogie cu cel al instrucțiunii **if**, cu deosebirea că instrucțiunea **if** permite **execuția** condiționată.
- Schema de principiu a modului de utilizare a celor patru directive este schițată în *slide*-ul următor.

```
#if conditie_1
    bloc_instructiuni_1
#elif conditie_2
    bloc_instructiuni_2
...
#elif conditie_n
    bloc_instructiuni_n
#else
    bloc_instructiuni_implicit
#endif
```

Aproape orice expresie ce are ca valoare o **constantă**. În general, constante declarate cu **#define**. Nu este permisă utilizarea operatorului `sizeof()`, a conversiilor de tip sau a tipului *float*.

Oricâte instrucțiuni C, de orice tip, inclusiv directive preprocesor.

Putem scrie oricâte **#elif** dorim, dar numai o singură **#else** !

Obligatorii ! Restul sunt opționale.

Efectul utilizării directivelor:

- Dacă valoarea expresiei **conditie_1** este “adevărat” (diferită de zero), se va compila **bloc_instructiuni_1**;
- În caz contrar, se testează în ordine condițiile asociate directivelor **#elif**. Se va compila blocul de instrucțiuni asociat primei directive **#elif** a cărei condiție are valoarea “adevărat” (diferită de zero);
- Dacă nici una dintre condițiile asociate directivelor **#elif** nu are valoarea “adevărat (diferită de zero), se va compila blocul de instrucțiuni asociat directivei **#else**.

Obs. Se compilează cel mult un bloc de instrucțiuni, dintre cele specificate. Dacă nu se utilizează directiva **#else**, este posibil chiar să nu se compileze nimic.

Exemple de utilizare

Scrierea unei aplicații care să fie folosită în mai multe țări ale lumii

Informațiile despre momentul de timp, data calendaristică, moneda, limba utilizată, unitățile de măsură pentru greutate și lungime etc. diferă, de multe ori, în funcție de țară.

Se pot scrie fișiere *header* cu informații specifice pentru fiecare țară, iar aplicația poate conține un fragment de program de tipul:

```
#if ENGLAND == 1
#include "england.h"
#elif FRANCE == 1
#include "france.h"
#elif ITALY == 1
#include "italy.h"
#else
#include "usa.h"
#endif
```

Folosind în aplicație directive **#define** se va putea stabili ce fișier *header* va fi inclus la momentul compilării. De exemplu, scriind **#define ITALY 1**, fișierul **italy.h** va fi cel inclus la compilare.

Scrierea în programele C a unor fragmente de cod specializate în testarea funcționării și depistarea eventualelor erori. Fragmente de program vor fi compilate și apoi utilizate în momentul execuției numai dacă programatorul setează în mod corespunzător valoarea unei constante.

De exemplu, se poate defini o constantă simbolică, numită **DEBUG**, ce va fi setată cu valoarea **1**, dacă dorim depanarea programului sau cu valoarea **0**, dacă nu dorim acest lucru. Setarea valorii se poate face prin scrierea în codul sursă a unei directive **#define DEBUG 1**. Apoi, oriunde și ori de câte ori considerăm necesar, putem scrie în programul sursă următorul tip de fragment de cod:

```
#if DEBUG == 1
```

Fragment de program C pentru testare-depistare de erori

```
#endif
```

În timpul fazei de testare a programului, se asigură astfel compilarea fragmentului de program specializat. După ce această fază s-a încheiat și există certitudinea că programul funcționează corect, se modifică valoarea constantei, scriind **#define DEBUG 0** și se recompilează programul, fără fragmentul respectiv.

Un exemplu concret, dar nu neapărat reprezentativ, ar putea fi:

```
#define DEBUG 1
```

```
...
```

```
puts("Tastati un nume:");
```

```
gets(num);
```

```
#if DEBUG == 1
```

```
    printf("Numele tastat a fost: %s\n", num);
```

```
#endif
```

```
...
```

Operatorul **defined()**

- Testează faptul că un anumit identificator este sau nu definit.
- Valoarea expresiei **defined(NUME)** este “adevarat”/“fals” dacă **NUME** este/nu este definit, indiferent de valoarea acestuia.

Exemplul anterior (depistarea de erori), se poate rescrie astfel:

```
#if defined( DEBUG )
```

Fragment de program C pentru depistare-tratare de erori

```
#endif
```

- Operatorul **defined()** poate fi folosit pentru a asocia o definiție unui nume, cu condiția ca numele să nu fi fost definit anterior. De exemplu:

↙ operator de negare logică

```
#if !defined( NR )
```

```
/* dacă NR nu e definit */
```

```
#define NR 30
```

```
/* definește NR ca fiind 30 */
```

```
#endif
```

Prevenirea includerii unui fișier header de mai multe ori în același program

Detalii despre modul în care poate să apară o astfel de situație - **în cadrul cap.12**

Exemplul următor presupune că dorim să scriem fișierul header numit **prog.h**, pe care îl vom edita în următoarea formă:

```
#if defined(PROG_H)
    /* înseamnă că fișierul prog.h a fost deja inclus! */
#else
    /* înseamnă că fișierul prog.h nu a fost încă inclus! */
    #define PROG_H
    /* aici se va scrie conținutul propriu-zis al fișierului header prog.h */
#endif
```

Drept urmare, constanta **PROG_H** (aleasă special, ca nume, asemănătoare numelui de fișier la care se referă) se definește doar la prima includere a fișierului în alt fișier (cu o directivă **#include**) și nu va mai permite apoi alte includeri similare, deoarece va rămâne definită.

11.4 Directiva #undef

Efect opus directivei **#define**: “rupe” asocierea numelui (identificatorului) cu valoarea specificată anterior printr-o directivă **#define**.

Reluând exemplul referitor la depistarea erorilor, se poate scrie:

#define DEBUG 1

/* în această parte a programului C, toate aparițiile identificatorului **DEBUG** vor fi înlocuite cu valoarea **1**, iar expresia **defined(DEBUG)**, dacă este utilizată, are valoarea “adevărat” */

#undef DEBUG

/* în această parte a programului C, aparițiile identificatorului **DEBUG** nu vor fi înlocuite, iar expresia **defined(DEBUG)**, dacă este utilizată, are valoarea “fals” */

Concluzie: directivele **#undef** și **#define** pot fi utilizate pentru crearea unui nume (identificator) definit numai în anumite zone ale programului. Acestea directive pot fi folosite și în combinație cu directiva **#if**.

Exemplificare pe calculator – conținutul fisierului stdio.h ...

11.5 Macro-instrucțiuni predefinite

Numeroase... Dintre ele (În scriere, se folosesc **câte 2 caractere de subliniere**, înainte și după literele ce specifică numele macro-instrucțiunii.):

- **__DATE__** și **__TIME__** sunt înlocuite cu data calendaristică și momentul de timp curente (cele la care este precompilat fișierul sursă). Aceste informații sunt deosebit de utile atunci când se lucrează cu diferite versiuni ale aceluiași program C. Prin afișarea lor la momentul execuției, se poate preciza versiunea de program utilizată în acel moment;
- **__LINE__** este înlocuită cu numărul liniei de program din fișierul sursă de care aparține;
- **__FILE__** este înlocuită cu numele fișierului sursă din care face parte.

Ultimele două macro-definiții menționate sunt foarte utile pentru depistarea și tratarea erorilor ce apar în cursul creării programelor C. De exemplu, dacă în fișierul sursă numit **aplicatie.C** există fragmentul de program:

```
10: ...  
11: printf( "Programul %s: (%d) Eroare la deschidere fisier ", __FILE__, __LINE__ );  
12: ...
```

pe ecranul monitorului se va afișa mesajul:

Programul aplicatie.C: (11) Eroare la deschidere fisier

Folosirea directivelor preprocesor ușurează în mod semnificativ activitatea de depanare și întreținere a programelor mari, formate din mai multe fișiere sursă și *header*, ce vor fi prezentate, pe scurt în cadrul capitolului 12.

11.6 Funcții cu număr variabil de parametri

Discutie despre functii obisnuite si functia de biblioteca *printf()*, cu vizualizarea prototipului din *stdio.h*

Detalii scrise si explicate in timpul cursului....

Instrumente necesare pentru scrierea unei functii cu numar variabil de parametri

- in fişierul header **stdarg.h**
- sunt folosite in functia cu numar variabil de parametri pentru a accesa valorile parametrilor

va_list	- tip de date pointer
va_start()	- macro-definitie folosita pentru initializarea listei de parametri (de argumente)
va_arg()	- macro-definitie folosita pentru extragerea din lista a valorii cate unui parametru (argument)
va_end()	- macro-definitie folosita pentru "curatenie" („clean up”) dupa extragerea din lista a valorii tuturor parametrilor

În scrierea funcției, trebuie parcurse următoarele etape:

1. Declararea unei variabile pointer de tip **va_list**. Va fi folosită pentru accesarea fiecărui parametru în parte. De cele mai multe ori i se dă numele **arg_ptr** dar acest lucru nu este obligatoriu!
2. Apelarea macro-ului **va_start()**, cu parametrii: pointerul **arg_ptr** și numele ultimului parametru „fix” al funcției. Macro-ul **va_start()** nu returnează nici o valoare ci doar inițializează pointerul **arg_ptr** astfel încât să indice primul parametru din lista celor „variabili”.
3. Obținerea valorii fiecărui parametru „variabil” prin apelarea succesivă a macro-ului **va_arg()** cu parametrii: pointerul **arg_ptr** și **tipul de data** corespunzător parametrului „variabil”. Valoarea returnată de **va_arg()** este **valoarea** acelui parametru. Dacă funcția are **n** parametri „variabili”, trebuie apelat **va_arg()** de **n** ori pt. a obține valorile tuturor parametrilor, în ordinea specificată prin scrierea lor în lista „variabila”.
4. După obținerea tuturor valorilor dorite, se apelează **va_end()**, cu parametru: pointerul **arg_ptr**. În anumite implementări, se realizează în acest fel anumite operații de „curățare” („clean up”).

Exemplu preluat din „Teach Yourself C in 21 Days”

Functie cu număr variabil de parametri ce calculeaza media aritmetica a unor valori numerice intregi.

Are un singur parametru „fix” ce este folosit pentru a preciza numarul de valori numerice intregi. Acestea, la randul lor, constituie lista de parametri „variabili” → → →

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

```
#include <stdarg.h>
```

```
float media (int numar, ...);    /* prototipul functiei cu numar variabil de parametri */
```

```
int main(void)
```

```
{ float x;
```

```
  x = media (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
/* se calculeaza media aritmetica a celor 10 valori */
```

```
  printf("\n Prima medie aritmetica este: %f.", x);
```

```
  x = media (5, 121, 206, 76, 31, 5);
```

```
/* se calculeaza media aritmetica a celor 5 valori */
```

```
  printf("\n A doua medie aritmetica este: %f.\n", x);
```

```
  return 0;
```

```
}
```

```
float media (int numar, ...)
```

```
{ /* Se declara o variabila de tip va_list. */
```

```
  va_list arg_ptr;
```

```
  int nr, total = 0;
```

```
  /* Se initializeaza pointerul la parametrii „variabili” */
```

```
  va_start(arg_ptr, numar);
```

```
  /* Se acceseaza, pe rand, parametrii din lista de parametri „variabili” */
```

```
  for (nr = 0; nr < numar; nr++)
```

```
    total += va_arg( arg_ptr, int );
```

```
  /* Se incheie lucrul cu elementele specifice din STDARG.H */
```

```
  va_end(arg_ptr);
```

```
  /* Se calculeaza media aritmetica a valorilor numerice intregi */
```

```
  return ( (float) total / numar);
```

```
}
```

Pe ecran:

Prima medie aritmetica este: 5.500000.

A doua medie aritmetica este: 87.800003.

Demonstratie: [Fct_var_1.c](#)

Alt exemplu, preluat din „*The C Programming Language*” - Brian W. Kernighan, Dennis M. Ritchie

Se definește o nouă funcție pentru afișarea pe ecran a unor valori, numită ***minprintf()***, cu număr variabil de parametri, asemănătoare cu un ***printf()*** minimal.

Noua funcție va fi capabilă să afișeze numai valori de tip întreg, real, șir de caractere (specificate cu format %d, %f, %s) și caractere „escape” (de exemplu, '\n')

→ → →

```

#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
void minprintf(char *fmt, ...)
{
    va_list ap;
    char *p, *sval;
    int ival;
    double dval;
    va_start(ap, fmt);
    for (p = fmt; *p; p++)
        { if (*p != '%')
            { putchar(*p);
              continue;
            }
          switch (*++p)
            { case 'd': ival = va_arg(ap, int);
              printf("%d", ival);
              break;
              case 'f': dval = va_arg(ap, double);
              printf("%f", dval);
              break;
              case 's': for (sval = va_arg(ap, char *); *sval; sval++)
                          putchar(*sval);
              break;
              default: putchar(*p);
                      break;
            }
          }
    va_end(ap);
}

```

```

int main (void)
{ int m=3;
  float x=25.4f;
  char ceva[]="mesaj";
  minprintf("intreg=%d, real=%f, sir=%s\n",m,x,ceva);
  minprintf("intreg=%d\n\t real=%f\n\t\t sir=%s\n",m,x,ceva);
  minprintf("Doar intreg=%d\n",m);
  getch();
  return 0;
}

```

Pe ecran:

```

intreg=3, real=25.400000, sir=mesaj
intreg=3
        real=25.400000
            sir=mesaj
Doar intreg=3

```

Demonstratie: Fct_var_2.c