

Programarea calculatoarelor

FMI

Secția Calculatoare și tehnologia informației, anul I

Cursul 4 / 28.10.2024

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: pointeri, tablouri, șiruri de caractere, **structuri, uniuni, câmpuri de biți, enumerări**
- **Instrucțiuni de control**
- **Directive de preprocesare. Macrodefiniții.**
- Funcții de citire/scriere.
- Etapele realizării unui program C.

❑ Fișiere text

- Funcții specifice de manipulare.

❑ Funcții (1)

- Declarare și definire. Apel. Metode de transmitere a paramerilor. Pointeri la funcții.

❑ Tablouri și pointeri

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

❑ Șiruri 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.

Cursul de azi

1. Tipuri derivate de date:

- a. structuri,**
- b. câmpuri de biți,**
- c. uniuni,**
- d. enumerări,**
- e. tipuri definite de utilizatori**

2. Instrucțiuni de control

3. Directive de preprocesare. Macrodefiniții.

Tipuri de date structurate

Limbajul C permite creare tipurilor de date în 5 moduri:

- a. **structura/ articol/ înregistrare (**struct**)** – grupează mai multe variabile sub același nume;
- b. **câmpul de biți (variațiune a structurii)** → acces ușor la biții individuali
- c. **uniunea (**union**)** – face posibil ca aceleași zone de memorie să fie definite ca două sau mai multe tipuri diferite de variabile
- d. **enumerarea (**enum**)** – listă de constante întregi cu nume
- e. **tip definit de utilizator (**typedef**)** – definește un nou nume pentru un tip existent

a. Structuri

❑ variabile grupate sub același nume.

❑ sintaxa:

```
struct <nume> {  
    < tip 1 >    <variabila 1>;  
    < tip 2 >    <variabila 2>;  
    -----  
    < tip n >    <variabila n>;  
} lista_identificatori_de_tip_struct;
```

❑ variabilele care fac parte din structură sunt denumite membri (**elemente** sau **câmpuri**) ai structurii.

Structuri

```
struct <nume> { < tip 1 >    <variabila 1>;  
               < tip 2 >    <variabila 2>;  
               -----  
               < tip n >    <variabila n>; } lista_identificatori;
```

Obs. 1:

- ✓ dacă numele structurii (<nume>) lipsește, structura se numește **anonimă**.
- ✓ Dacă lista identificatorilor declarați lipsește, se definește doar tipul structură.
- ✓ Cel puțin una dintre aceste specificații trebuie să existe.

Obs. 2: dacă <nume> este prezent → se pot declara noi variabile de tip structură:

struct <nume> <lista noilor identificatori>;

Obs. 3: referirea unui membru al unei variabile de tip structură → **operatorul de selecție punct .** care precizează identificatorul variabilei și al câmpului.

```
#include <stdio.h>
struct Data
{int zi, luna, an;};
Data azi={27,10,2024};
int main()
{
    printf("azi este %d.%d.%d \n", azi.zi, azi.luna, azi.an);
    return 0;
}
```

ERROR!

```
/tmp/9z8aqffi14.c:4:1: error: unknown type name 'Data'; use 'struct' keyword to refer to
the type
4 | Data azi={27,10,2024};
  | ^~~~
  | struct
/tmp/9z8aqffi14.c:4:14: warning: excess elements in scalar initializer
4 | Data azi={27,10,2024};
```

```
1 #include <stdio.h>
2 struct Data
3     {int zi, luna, an;};
4 struct Data azi={27,10,2024};
5 int main()
6 {
7     printf("azi este %d.%d.%d \n", azi.zi, azi.luna, azi.an);
8     return 0;
9 }
```

azi este 27.10.2024

=== Code Execution Successful ===

```

1  #include <iostream>
2  using namespace std;
3
4  struct Data
5      {int zi, luna, an;};
6  Data azi={27,10,2024};
7  int main()
8  {
9      cout<< azi.zi<<'.'<< azi.luna<<'.'<< azi.an;
10     return 0;
11 }

```

27.10.2024

=== Code Execution Successful ===

```

1  #include <stdio.h>
2  struct Data
3      {int zi, luna, an;};
4  struct Data azi={27,10,2024};
5  int main()
6  {
7      printf("azi este %d.%d.%d \n", azi.zi, azi.luna, azi.an);
8      return 0;
9  }

```

azi este 27.10.2024

=== Code Execution Successful ===

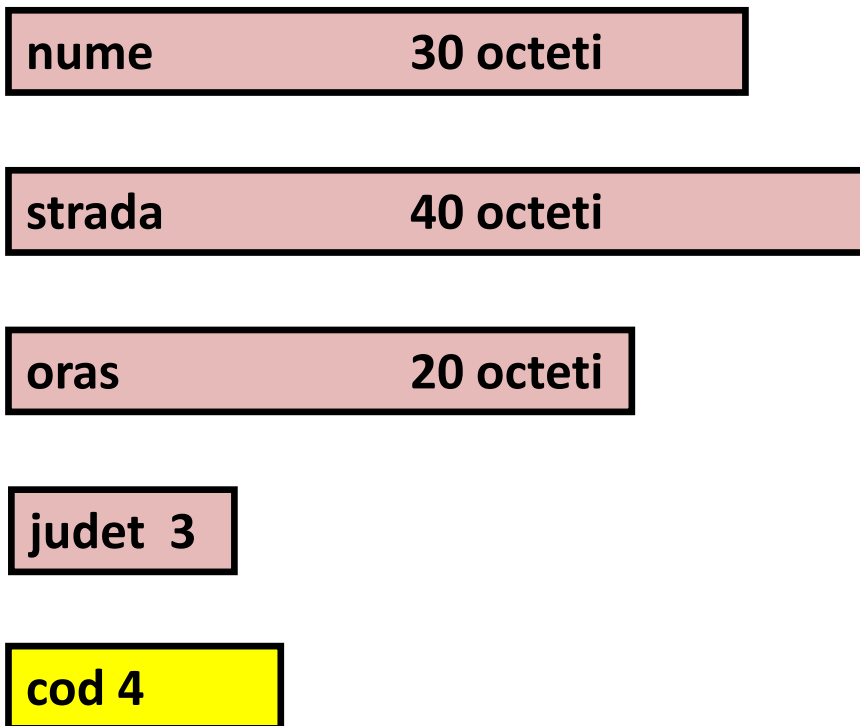
Structuri. Exemplu

```
struct adrese {  
    char nume[30];  
    char strada[40];  
    char oras[20], judet[3];  
    int cod;;  
}
```

struct adrese A;

- ❑ numele **adrese** identifică această structură de date particulară
- ❑ **struct adrese A;** declară o variabilă de tip adrese și îi alocă memorie

Structura din memorie pentru variabila A de tip adrese



Structuri. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      struct adrese {
6          char nume[30];
7          char strada[40];
8          char oras[20], judet[3];
9          int cod;
10     };
11     struct adrese A;
12
13     printf("%d\t", (int)sizeof(A.nume));
14     printf("%d\t", (int)sizeof(A.cod));
15     printf("%d", (int)sizeof(A));
16
17     return 0;
18 }
```

Ce afișează programul?

30 4 100

*Compilerul alocă
memorie în plus pentru
alinieare (multiplu de 4)*

Structuri. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      struct adrese {
6          char nume[30];
7          char strada[40];
8          char oras[20], judet[6];
9          int cod;
10     };
11     struct adrese A;
12
13     printf("%d\t", (int)sizeof(A.nume));
14     printf("%d\t", (int)sizeof(A.cod));
15     printf("%d", (int)sizeof(A));
16
17     return 0;
18 }
```

Ce afișează programul?

30

4

100

Structuri. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      struct adrese {
6          char nume[30];
7          char strada[40];
8          char oras[20], judet[3];
9      };
10     struct adrese A;
11
12     printf("%d\t", (int)sizeof(A.nume));
13     printf("%d\t", (int)sizeof(A));
14
15     return 0;
16 }
```

Ce afișează programul?

30

93

Compilerul alocă memorie în plus pentru aliniere numai când avem tipuri de date diferite.

Structuri. Exemplu

```
struct adrese {  
    char nume[30];  
    char strada[40];  
    char oras[20];  
    char judet[3];  
    int cod;  
} A, B, C;
```

- Definește un tip de structură numit **adrese**
- Declară ca fiind de acest tip variabilele **A, B, C**

```
struct {  
    char nume[30];  
    char strada[40];  
    char oras[20];  
    char judet[3];  
    int cod;  
} A;
```

- Declară o variabilă numită **A** definită de structura care o precede.

Structuri. Exemplu

```
struct adrese {  
    char nume[30];  
    char strada[40];  
    char oras[20];  
    char judet[3];  
    int cod;  
} A, B, C;
```

- Definește un tip de structură numit **adrese**
- Declară ca fiind de acest tip variabilele **A, B, C**

❑ **accesul la membrii structurii**
se face prin folosirea
operatorului punct:

nume_variabila.nume_camp

❑ **citirea:**

scanf("%d", &C.cod);

❑ **atribuiri pentru variabile de tip structură:**

B = A;

Obs. Singura operație cu structuri este atribuirea.

b. Câmpuri de biți

- ❑ tip special de membru al unei structuri care definește cât de lung trebuie să fie câmpul, în biți;
- ❑ permite accesul la un singur bit;
- ❑ putem stoca mai multe variabile boolene într-un singur octet;
- ❑ nu se poate obține adresa unui câmp de biți;
- ❑ **adaugă mai multă structurare.**

❑ **Sintaxa:**

```
struct <nume> {  
    < tip 1 >          <variabila 1>: lungime;  
    < tip 2 >          <variabila 2>: lungime;  
    -----  
    < tip n >          <variabila n>: lungime;  
} lista_identificatori_de_tip_struct;
```

Câmpuri de biți

Sintaxa:

```
struct <nume> {  
    < tip 1 >          <variabila 1>: lungime;  
    < tip 2 >          <variabila 2>: lungime;  
    -----  
    < tip n >          <variabila n>: lungime;  
} lista_identificatori_de_tip_struct;
```

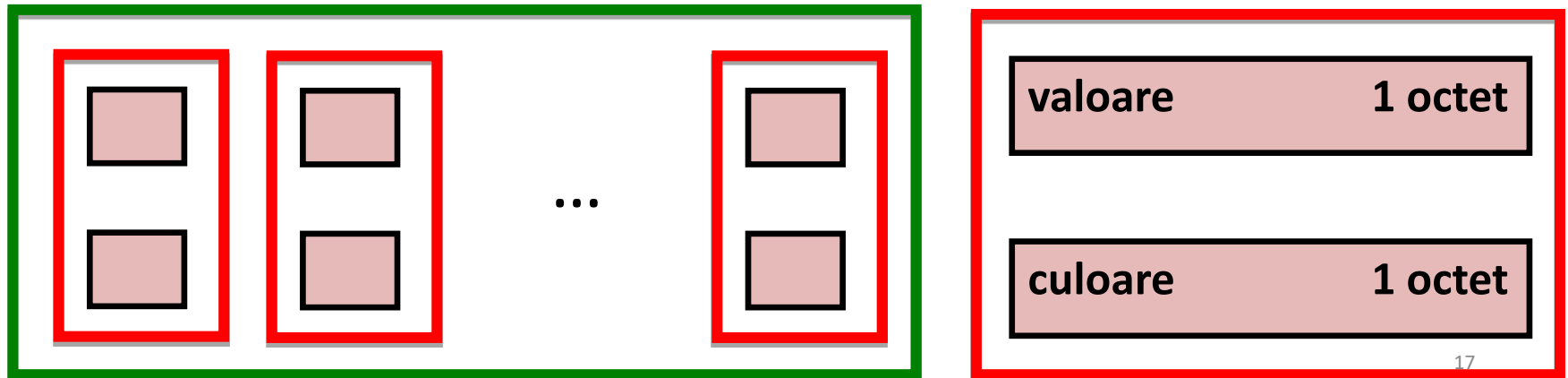
Observații:

- ❑ tipul câmpului de biți poate fi doar: **int**, **unsigned** sau **signed**;
- ❑ câmpul de biți cu lungimea 1 → **unsigned** (un singur bit nu poate avea semn);
- ❑ unele compilatoare → **doar unsigned**;
- ❑ lungime → **numărul de biți dintr-un câmp**.

Câmpuri de biți. Exemplu

Joc de cărți: reprezentăm o carte printr-o structură:

```
struct carteJoc {  
    unsigned char valoare; // valori între 1 și 13  
    unsigned char culoare; // valori între 1 și 4  
};  
struct carteJoc PachetCartiJoc[52];
```



Câmpuri de biți. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      struct carteJoc{
6          unsigned char valoare;
7          unsigned char culoare;
8      }A;
9      struct carteJoc PachetCartiJoc[52];
10
11     printf("%d\t", (int)sizeof(A));
12     printf("%d\t", (int)sizeof(PachetCartiJoc));
13
14     return 0;
15 }
```

Ce afișează programul?

2 104

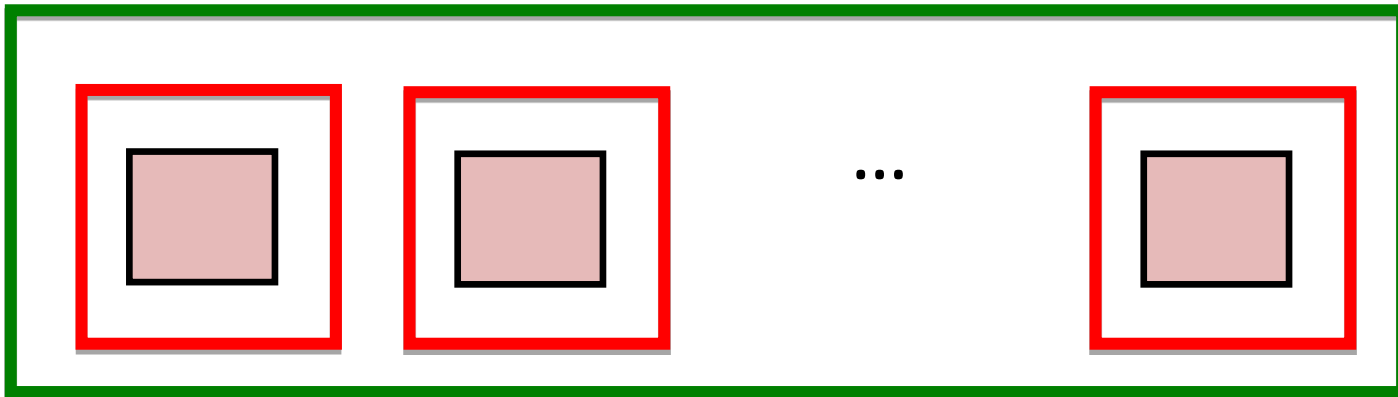
Dimensiune A este 2

Dimensiune PachetCartiJoc este 104(=2*52)

Câmpuri de biți. Exemplu

Joc de cărți: reprezentăm o carte printr-o structură:

```
struct carteJoc {  
    unsigned char valoare: 4; // 4 biți  
    unsigned char culoare: 2; // 2 biți  
};  
struct carteJoc PachetCartiJoc[52];
```



Câmpuri de biți. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      struct carteJoc{
6          unsigned char valoare:4;
7          unsigned char culoare:2;
8      }A;
9      struct carteJoc PachetCartiJoc[52];
10
11     printf("%d\t", (int)sizeof(A));
12     printf("%d\t", (int)sizeof(PachetCartiJoc));
13
14     return 0;
15 }
```

Ce afișează programul?

1 52

Dimensiune A este 1

Dimensiune PachetCartiJoc este 52

Obs: Folosește doar un octet pentru a păstra două informații: valoare și culoare

Câmpuri de biți. Exemplu

□ în aceeași structură putem combina câmpuri de biți și membri normali

```
struct adrese {  
    char nume[30];  
    char strada[40];  
    char oras[20];  
    char jud[3];  
    int cod;  
};
```

```
struct angajat {  
    struct adrese A;  
    float plata;  
    unsigned statut: 1; // activ sau intrerupt  
    unsigned plata_oras: 1; // plata cu ora  
    unsigned impozit: 3; // impozit rezultat  
};
```

Obs: definește o înregistrare despre salariat care *folosește doar un octet pentru a păstra 3 informații*: statutul, dacă este platit cu ora și impozitul.

□ fără câmpul de biți, aceste informații ar fi ocupat 3 octeți.

c. Uniuni

- ❑ tip special de structuri ai cărei membri folosesc la momente diferite aceeași locație în memorie;
- ❑ membrii unei uniuni au de obicei tipuri diferite.

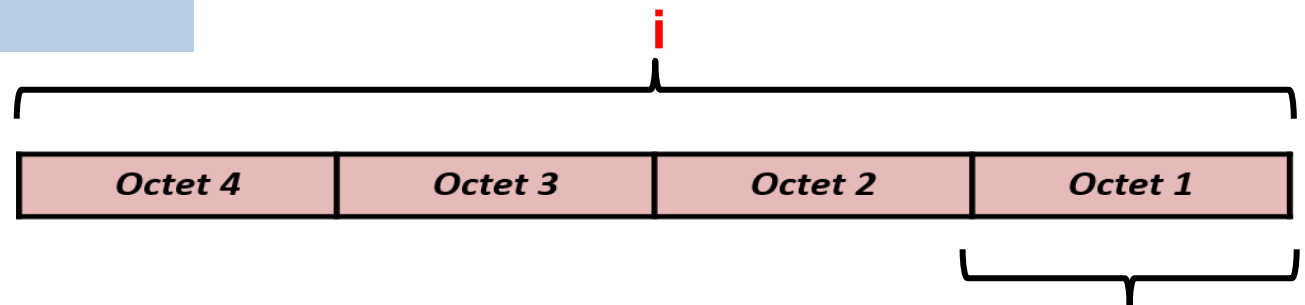
Sintaxa:

```
union <nume> {  
    < tip 1 >  <variabila 1>;  
    < tip 2 >  <variabila 2>;  
    -----  
    < tip n >  <variabila n>;  
} lista_identificatori_tip_union;
```

Uniuni. Exemplu

- ❑ tip special de structuri ai cărei membri folosesc la momente diferite aceeași locație în memorie;
- ❑ membrii unei uniuni au de obicei tipuri diferite;
- ❑ când este declarată o variabilă de tip **union** compilatorul alocă automat memorie suficientă pentru a păstra cel mai mare membru al acesteia

```
union tip_u {  
    int i;  
    char ch;};  
union tip_u A;
```



Uniuni. Exemplu

main.c

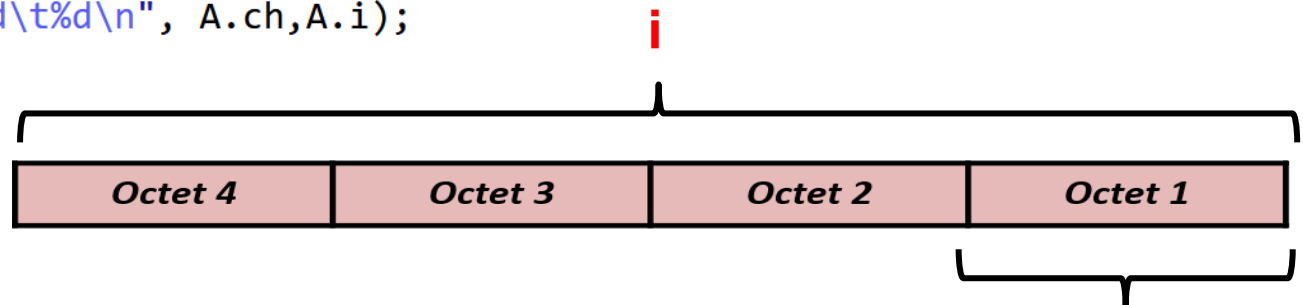
```
1 #include <stdio.h>
2
3 int main()
4 {
5     union tip_u {
6         int i;
7         char ch;
8     };
9     union tip_u A;
10    printf("%d\n", (int)sizeof(A));
11
12    A.ch=10;
13    printf("%d\t%d\n", A.ch,A.i);
14
15    A.i=296;
16    printf("%d\t%d\n", A.ch,A.i);
17
18    return 0;
19 }
```

Ce afișează programul?

4

10 10

40 296



Explicație

Orice număr mai mare de 127 se memorează pe mai mult de 8 biți.
Orice număr mai mic de -128 se memorează pe mai mult de 8 biți.

$$10_{(10)} = 2^3 + 2 = 0000 \ 1010_{(2)}$$

$$296_{(10)} = 2^8 + 2^5 + 2^3 = \textcolor{red}{1} \ 00\textcolor{blue}{10} \ 1000_{(2)} \longrightarrow 00\textcolor{blue}{10} \ 1000_{(2)} = 2^5 + 2^3 = 32 + 8 = 40_{(10)}$$

$$296:2=148+0$$

$$148:2=74+0$$

$$74:2=37+0$$

$$37:2=18+1$$

$$18:2=9+0$$

$$9:2=4+1$$

$$4:2=2+0$$

$$2:2=1+0$$

$$1:2=0+1$$

d. Enumerări

- ❑ mulțime de constante de tip întreg care specifică toate valorile permise pe care le poate avea o variabilă de acel tip
- ❑ atât numele generic al enumerării cât și lista de variabile sunt optionale
- ❑ constanta unui element al enumerării este fie asociată implicit, fie explicit. Implicit, primul element are asociată valoarea 0, iar pentru restul este valoarea precedentă+1.

Sintaxa:

```
enum <nume> {  
    lista enumerarilor  
} lista variabile;
```

Enumerări. Exemplu

`enum {a, b, c, d};` → a = 0, b = 1, c = 2, d = 3

`enum {a, b, c=7, d};` → a = 0, b = 1, c = 7, d = 8

`enum {a=4, b=-3, c=9, d=-8};`

main.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     enum zile {luni, marti, miercuri, joi, vineri, sambata, duminica};
6
7     printf("\tAzi este ziua cu numarul %d din saptamana.\n", joi);
8
9     return 0;
10 }
```

input

Azi este ziua cu numarul 3 din saptamana.

Enumerări. Exemplu

`enum {a, b, c=7, d};` $\rightarrow a = 0, b = 1, c = 7, d = 8$

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      enum zile {luni=1, marti, miercuri, joi, vineri, sambata, duminica};
6
7      printf("\tAzi este ziua cu numarul %d din saptamana.\n", joi);
8
9      return 0;
10 }
```

<



input

Azi este ziua cu numarul 4 din saptamana.

Enumerări. Exemplu

main.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5  enum zile {luni=1, marti, miercuri,
6             joi, vineri, sambata, duminica}azi, maine;
7  azi=joi;
8  printf("\tAzi este ziua cu numarul %d din saptamana.\n", azi);
9  maine=azi+1;
10 printf("\tMaine este ziua cu numarul %d din saptamana.\n", maine);
11 return 0;
12 }
```

input

```
Azi este ziua cu numarul 4 din saptamana.
Maine este ziua cu numarul 5 din saptamana.
```

e) Specificatorul typedef

- ❑ definește explicit noi tipuri de date.
- ❑ nu se declară o variabilă sau o funcție de un anumit tip, ci se asociază un nume (sinonimul) tipului de date.

- ❑ **sintaxa:**

typedef <definiție tip> <identificator>;

- ❑ **exemple:**

typedef unsigned int **natural**;

typedef long double **tablouNumereReale [100]** ;

tablouNumereReale a, b, c;

natural m, n, i;

Cursul de azi

1. Tipuri derivate de date: structuri, uniuni, câmpuri de biți, enumerări, tipuri definite de utilizatori
2. Instrucțiuni de control
3. Directive de preprocesare. Macrodefiniții.

Instrucțiuni de control

□ reprezintă:

- elementele fundamentale ale funcțiilor
- comenzile date calculatorului
- determină fluxul de control al programului (ordinea de execuție a operațiilor din program)

□ instrucțiuni de bază

1. instrucțiunea expresie
2. instrucțiunea vidă
3. instrucțiuni secvențiale/liniare
4. instrucțiuni decizionale/selective simple sau multiple
5. instrucțiuni repetitive/ciclice/iterative
6. instrucțiuni de salt condiționat/necon condiționat
7. instrucțiunea return

Instrucțiuni de control

- instrucțiuni compuse

- create prin combinarea instrucțiunilor de bază

- programare structurată

- **Teorema Böhm-Jacopini:** fluxul de control poate fi exprimat folosind doar trei tipuri de **instrucțiuni de control:**

- instrucțiuni **secvențiale**
 - instrucțiuni **decizionale**
 - instrucțiuni **repetitive**

1. Instrucțiunea expresie

- ❑ formată dintr-o expresie urmată de semnul ;
- ❑ 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;
```

2. 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 acolade, { }
 - astfel compilatorul va trata secvența de instrucțiuni ca pe o singură instrucțiune
 - *{secvență de declarații și instrucțiuni }*

3. 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. Exemplu

Enunț: Se citesc numerele naturale a și b de la tastatură. Să se afișeze ultima cifră a numărului a^b .

```
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;
26 }
27
```

Instrucțiunea IF. Erori frecvente:

❑ neincluderea acoladelor:

```
if (a>b)
```

```
    a=a+b;
```

```
    b=a;           //întotdeauna se va executa instrucțiunea 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");
```

- ❑ mesajul *a este 10* nu va fi afișat
 - ❑ după testarea egalității folosind operatorul == se returnează 0 (2 nefiind egal cu 10)

- ❑ mesajul *a este 10* va fi afișat întotdeauna
 - ❑ expresia *a = 10*
 - ❑ a ia valoarea 10
 - ❑ se evaluează adevărat la executarea instrucțiunii printf

Instrucțiuni IF imbricate

- ❑ o instrucțiune if poate conține alte instrucțiuni if pe oricare ramură
- ❑ 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");
```

Instrucțiuni 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: EXCELENT");  
    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

Instrucțiunea 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. Exemplu

Enunț: Se citesc numerele naturale a și b de la tastatură. Să se afișeze ultima cifră a numărului a^b .

```
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); 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 șiruri 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 acolade, deși pot fi mai multe instrucțiuni, iar ultima instrucțiune este de regulă instrucțiunea `break`.

Instrucțiunea SWITCH

- mod de funcționare și constrângeri (continuare):
 - 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 iese 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] !");
}

```

Instrucțiunea SWITCH

```
seminar1_2.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     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 }
```

Ce afișează programul?

12 33
2
4
8

5. 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 iterație
- ❑ instrucțiunile iterative furnizate de limbajul C sunt:
 - a) instrucțiunea repetitivă cu testare inițială **while**
 - b) instrucțiunea repetitivă cu testare finală **do-while**
 - c) instrucțiunea repetitivă cu testare inițială **for**

Instrucțiunea WHILE

- ❑ execută în mod repetat o instrucțiune atâta timp cât expresia de control este evaluată la adevărat
- ❑ evaluarea se efectuează la începutul instrucțiunii
 - ❑ dacă rezultatul corespunde valorii logice adevărat:
 - ❑ se execută corpul 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}
```

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 [X]
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
```

Observație

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

Instrucțiunea WHILE

Ce afișează următoarea secvență de cod?

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

CICLEAZĂ!

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

3 2 1

3 2 1 0

Instrucțiunea DO - WHILE

- ❑ efectuează în mod repetat o instrucțiune atâta 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

Instrucțiunea 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);
```

Rezultatul unei rulări a acestui program este:

```
    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;
}
```

```
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 iterații
- ❑ 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);
```

Instrucțiunile break, continue și goto

- realizează salturi

- întrerup 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 iterația imediat următoare în instrucțiunea repetitivă

- instrucțiunea goto produce un salt la o etichetă predefinită în cadrul aceleiași funcții

Instrucțiunea goto

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

```
    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:

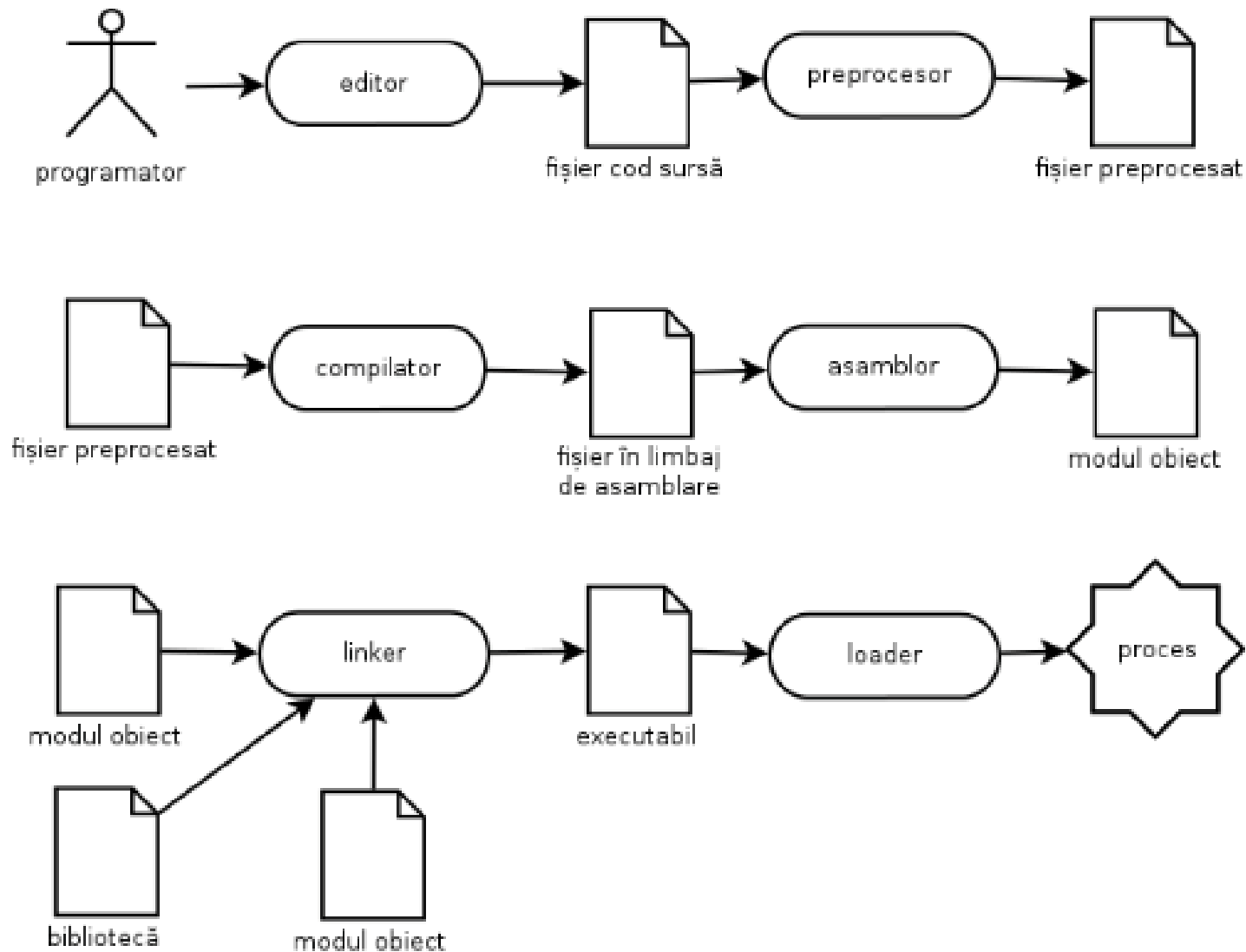
- `return;`
- `return expresie;`

```
seminar1.c 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
```

Cursul de azi

1. Tipuri derivate de date: structuri, uniuni, câmpuri de biți, enumerări, tipuri definite de utilizatori
2. Instrucțiuni de control
3. Directive de preprocesare. Macrodefiniții.

Etapele realizării unui program în C



Etapele realizării unui program în C

❑ etape pentru obținerea unui cod executabil:

1. **editarea** codului sursă

- ❑ salvarea fișierului cu extensia `.c`

2. **preprocesarea**

- ❑ efectuarea directivelor de preprocesare (**#include**, **#define**)
- ❑ ca un editor – modifică și adaugă la codul sursă

3. **compilarea**

- ❑ verificarea sintaxei
- ❑ codul este tradus din cod de nivel înalt în limbaj de asamblare

4. **asamblarea**

- ❑ transformare în cod obiect (limbaj mașină) cu extensia `.o`, `.obj`
- ❑ nu este încă executabil !

5. **link-editarea** (editarea legăturilor)

- ❑ combinarea codului obiect cu alte coduri obiect (al bibliotecilor asociate fișierelor header)
- ❑ transformarea adreselor simbolice în adrese reale

Preprocesarea în limbajul C

- 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 macrouri (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 #include. Exemple

```
// include fisierul stdio.h  
// din directoarele standard  
#include <stdio.h>
```

```
// include fisierul Liste.h;  
// cautarea se face intai in directorul  
// curent si dupa aceea in directoarele standard  
#include "Liste.h"
```

```
// include fisierul Masive.cpp din directorul  
// c:\Biblioteci; daca fisierul nu exista nu  
// mai este cautat in alta parte si se genereaza  
// o eroare de compilare  
#include "C:\Biblioteci\Masive.cpp"
```

Directiva **#define**

- ❑ folosită pentru definirea (înlocuirea) constantelor simbolice și a macrourilor
- ❑ 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 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

Directivele #define și #include. Exemplu

Ce afișează programul?

main.c	ceva.txt
1 #include <stdio.h>	
2 #include "ceva.txt"	
3 #define MIN 0	
4	
5 int main()	
6 {	
7 int a,b;	
8 do	
9 {	
10 printf("a="); scanf("%d", &a);	
11 printf("b="); scanf("%d", &b);	
12 }while (a<=MIN b<=MIN);	
13 printf("cmmdc(%d,%d)= %d\n",a,b,divizor(a,b));	
14 return 0;	
15 }	

main.c	ceva.txt
1	int divizor(int a, int b)
2	{
3	int c=a%b;
4	while(c)
5	{ a=b; b=c; c=a%b;}
6	return b;
7	}

Directiva #define. Exemplu

```
#include <stdio.h>
// definire parametru
#define DIM_VECTOR 20
// definire tip vector
#define TIP double
// definire mesaj
#define MESAJ "Calcul suma"
// definire cod pe mai multe linii
#define SEPARATOR printf( \
    "-----" \
    "\n");
TIP suma(TIP v[], int n){
    //validare lungime
    if (n > DIM_VECTOR)
        printf("dimensiunea este mai mare ca
        DIM_VECTOR\n");
    //calcul suma
    TIP suma = 0;
```

```
    for (int i = 0; i < n; i++)
        suma += v[i];
    return suma;}

int main()
{
    /*declarare vector folosind simbolurile
    specificate*/
    TIP v[DIM_VECTOR] = {1.1, 3.23, 6.62};
    int n = 3;
    //utilizare simboluri pentru mesaj
    MESAJ;
    //inserare cod prin preprocesor
    SEPARATOR;
    //apel functie
    printf("%f",suma(v, 3));
    return 0;
}
```

Directiva **#define**

- ❑ 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**

Typedef

- asociază un nume unui tip de date, predefinit sau definit de programator

- Sintaxa:

```
typedef descriere_tip nume_tip;
```

- Exemple:

```
typedef float real;
```

```
typedef int* PInt;
```

```
typedef int vector[20];
```

```
typedef struct {
```

```
    int grad; double coeficient[100];} polinom;
```

#define vs typedef

#define	typedef
<ul style="list-style-type: none">- Poate să definească aliasuri și pentru valori, de exemplu: 1 ca ONE, 3.14 ca PI etc	<ul style="list-style-type: none">- dă nume simbolice numai tipurilor. Definește un nou tip prin copierea și lipirea definițiilor campurilor unui struct
<ul style="list-style-type: none">- instrucțiunile #define sunt interpretate de preprocesor	<ul style="list-style-type: none">- instrucțiunile typedef sunt interpretate de compilator
<ul style="list-style-type: none">- Nu trebuie să existe punct și virgulă la sfârșitul lui #define	<ul style="list-style-type: none">- Se scrie punct și virgulă la sfârșitul lui typedef

Directiva **#define**

- definirea unui **macro**:

#define nume (lista-parametri) text

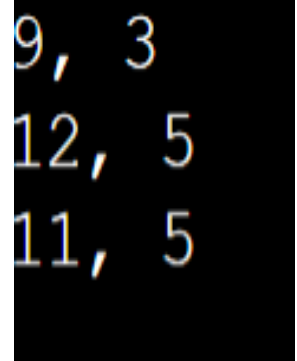
- numele macro-ului este **nume**
- lista de parametri este de ex.: **p1, p2, ..., pn**
- textul substituit este **text**
- parametrii formali sunt substituiți de cei actuali în **text**
- apelul macro-ului este similar apelului unei funcții
nume (p_actual1, p_actual2, ..., p_actualln)

Directiva #define

main.c

```
1  #include <stdio.h>
2  #define SQUARE(x) x*x
3
4  int main()
5  {
6      int a=3;
7      printf("%d, %d\n", SQUARE(a), a);
8      printf("%d, %d\n", SQUARE(a++), a);
9      printf("%d, %d\n", SQUARE(a+1), a);
10
11     return 0;
12 }
```

Ce afișează programul?



```
9, 3
12, 5
11, 5
```

Explicația în următorul diapozitiv.

Directiva #define

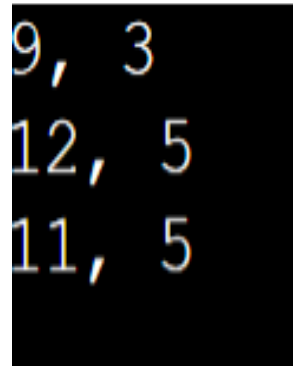
```
#define SQUARE(x)    x * x
```

- Dacă $x=3$ și apelăm: **SQUARE(x)**, atunci in program preprocesorul substituie: **3 * 3**
- Dacă $x=3$ și apelăm: **SQUARE(x++)**, atunci:

$x++ * (x++)=3*4=12$, iar $x=5$

- Dacă $x=5$ și apelăm: **SQUARE(x+1)**:

$x+1 * (x+1)=5+1*5+1=11$



9, 3
12, 5
11, 5

Directiva #define

Corecție:

```
#define SQUARE(x)  (x) * (x)
```

Când apelăm:

```
printf(“%d \n”, SQUARE( a + 1 ) )
```

preprocesorul înlocuiește:

```
printf(“%d \n”, ( a + 1 )*( a + 1 ) )
```

Directiva #define

main.c

```
1 #include <stdio.h>
2 #define SQUARE(x) x*x
3 #define Square(x) (x)*(x)
4 int main()
5 {
6     int a=3;
7     printf("%d, %d\n", SQUARE(a), a);
8     printf("%d, %d\n", SQUARE(a++), a); //a++*(a++)
9     printf("%d, %d\n", SQUARE(a+1), a); //a+1*a+1
10    int b=3;
11    printf("%d, %d\n", Square(b), b);
12    printf("%d, %d\n", Square(b++), b); //b++*(b++)
13    printf("%d, %d\n", Square(b+1), b); //(b+1)*(b+1)
14
15    return 0;
16 }
```

Ce afișează programul?

```
9, 3
12, 5
11, 5
9, 3
12, 5
36, 5
```

Directiva #define

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

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

```
{
```

```
    int a=3;
```

```
    printf("%d, %d\n",DOUBLE(a),a);
```

```
    printf("%d, %d\n",10*DOUBLE(a),a);
```

```
    printf("%d, %d\n",10*Double(a),a);
```

```
    printf("%d, %d\n\n",10*DoubleE(a),a);
```

```
    return 0;}
```

Ce afișează programul?

Directiva #define

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

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

```
{
```

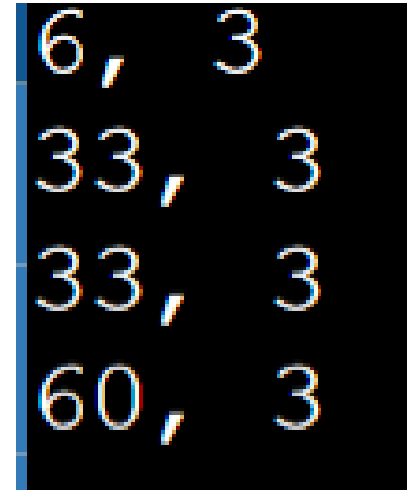
```
    int a=3;
```

```
    printf("%d, %d\n",DOUBLE(a),a); //a+a
```

```
    printf("%d, %d\n",10*DOUBLE(a),a); //10*a+a
```

```
    printf("%d, %d\n",10*Double(a),a); //10*a+a
```

```
    printf("%d, %d\n\n",10*DoubleE(a),a); //10*(a+a)
```



6,	3
33,	3
33,	3
60,	3

Directiva #define

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

Ce afișează programul?

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

```
{
```

```
    int a=3;
```

```
    printf("%d, %d\n", 10*DOUBLE(a+1), a);
```

```
    printf("%d, %d\n", 10*Double(a+1), a);
```

```
    printf("%d, %d\n\n", 10*DoubleE(a+1), a);
```

Directiva #define

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

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

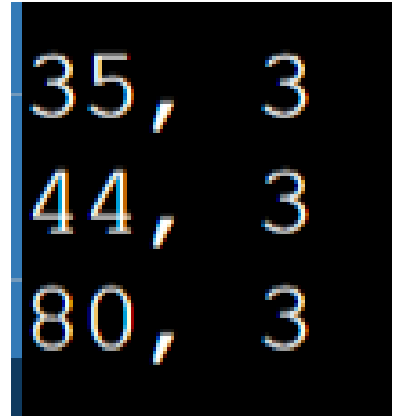
```
{
```

```
    int a=3;
```

```
    printf("%d, %d\n", 10*DOUBLE(a+1), a); //10*a+1+a+1
```

```
    printf("%d, %d\n", 10*Double(a+1), a); //10*(a+1)+(a+1)
```

```
    printf("%d, %d\n\n", 10*DoubleE(a+1), a); //10*(a+1+a+1)
```



35,	3
44,	3
80,	3

Directiva #define

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

Ce afișează programul?

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

```
{
```

```
    int a=3;
```

```
    printf("%d, %d\n", 10*DOUBLE(a++), a);
```

```
    a=3; printf("%d, %d\n", 10*Double(a++), a);
```

```
    a=3; printf("%d, %d\n\n", 10*DoubleE(a++), a);
```

Directiva #define

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

```
#define DOUBLE(x) x+x
```

```
#define Double(x) (x)+(x)
```

```
#define DoubleE(x) (x+x)
```

```
int main()
```

```
{ int a=3;
```

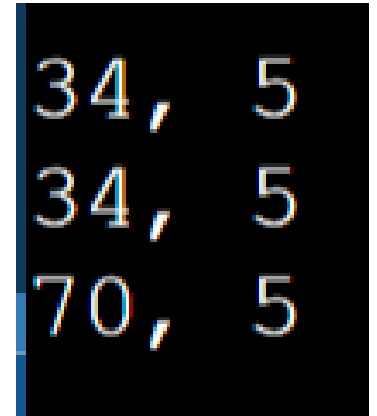
```
printf("%d, %d\n",10*DOUBLE(a++),a); //10*a++ +(a++)
```

```
a=3;
```

```
printf("%d, %d\n",10*Double(a++),a); //10*(a++)+(a++)
```

```
a=3;
```

```
printf("%d, %d\n\n",10*DoubleE(a++),a); //10*((a++)+(a++))
```



```
34, 5
34, 5
70, 5
```


Directiva #define

```
#define DOUBLE(x)  ( x ) + ( x )
```

```
a = 5;
```

```
printf(“%d \n”, 10 * DOUBLE( a ) );
```

//Echivalent cu:

```
a = 5;
```

```
printf(“%d \n”, 10 * ( a ) + ( a ) ); // => 55
```

Corectie:

```
#define DOUBLE(x)  ( ( x ) + ( x ) )
```

OBS. *Toate macro-urile care evaluează expresii numerice necesită parantezare în aceasta manieră pentru a evita interacțiuni nedorite cu alți operatori*

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 utilizarea doar pentru calcule simple
 - ❑ parametrul formal este înlocuit cu textul corespunzător parametrului actual, corespondența fiind pur pozițională
- ❑ timpul de procesare este mai scurt când se utilizează macro-uri (apelul funcției necesită timp suplimentar)

Directiva #define. Exemplu

main.c

```
1  #include <stdio.h>
2
3  //constante simbolice
4  #define ALPHA 30
5  #define BETA ALPHA+10
6  #define GAMMA (ALPHA+10)
7
8  //macro-uri
9  #define MIN(a,b) ((a<b)? a:b)
10 #define ABS1(x) (x<0)?-x:x
11 #define ABS2(x) (((x)<0)?(-x):(x))
12 #define INTERSCHIMB(tip,a,b)\
13     {tip c; c=a;a=b;b=c;}
14
```

OBS. Cu paranteză
vs fără paranteză

```
70, 80
70, -30
-90, 30
-90, 30
-90, -90, 40
40, -90
```

main.c

```
14
15 int main()
16 {
17     int x=2*BETA, y=2*GAMMA;
18     printf("%d, %d\n",x,y);
19     int m=MIN(x,y);
20     printf("%d, %d\n",m,BETA-x);
21     int A=ABS1(BETA-x);
22     int A1=ABS1((BETA-x));
23     printf("%d, %d\n",A,A1);
24     int B=ABS2(BETA-x);
25     int B2=ABS2((BETA-x));
26     printf("%d, %d\n",B,B2);
27     int C=ABS2((x-ALPHA));
28     printf("%d, %d, %d\n",A,B,C);
29     INTERSCHIMB(int,A,C);
30     printf("%d, %d\n",A,C);
31
32     return 0;
33 }
```

Directiva #define. Exemplu

main.c

```
1  #include <stdio.h>
2  // definire tip date
3  #define TIP int
4
5  // definire cod pe mai multe linii
6  #define DEBUG_PRINT printf( \
7      "File %s, line %d:\n"\
8      "x = %d, y = %d, z = %d ", \
9      __FILE__ , __LINE__ ,\      //2 macro-uri predefinite
10     x,y,z)
11
12  int main()
13  {
14      TIP x=3,y=5,z;
15      x *=2;
16      y += x;
17      z = x * y;
18      DEBUG_PRINT;
19  return 0;
20 }
```

```
File main.c, line 18:
x = 6, y = 11, z = 66
```

Example

```
#include <stdio.h>

#define medie(a,b,c) {\
float m = 0; \
m = 0.4 * (a + b) + 0.2 *c \
}

int main()
{
    int x = 2, y = 3, z = 4;

    medie(x,y,z);

    printf("medie = %f ",m);

    return 0;
}
```

```
#include <stdio.h>

#define medie(a,b,c) \
float m = 0; \
m = 0.4 * (a + b) + 0.2 *c \

int main()
{
    int x = 2, y = 3, z = 4;

    medie(x,y,z);

    printf("medie = %f ",m);

    return 0;
}
```

note: in expansion of macro 'medie'

error: 'm' undeclared (first use in this function)

note: each undeclared identifier is reported only once for each function it app

=== Build failed: 2 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===

```
medie = 2.800000
Process returned 0 (0x0)
```

Example

```
#include <stdio.h>

#define medie(a,b,c) 0.4 * (a + b) + 0.2 *c

int main()
{
    int x = 2, y = 3, z = 4;

    printf("medie = %f ",medie(x,y,z));

    return 0;
}
```

```
#include <stdio.h>

#define medie(a,b,c) \
float m = 0; \
m = 0.4 * (a + b) + 0.2 *c \

int main()
{
    int x = 2, y = 3, z = 4;

    medie(x,y,z);

    printf("medie = %f ",m);

    return 0;
}
```

Macro versus functii

Avantaj macro-uri:

Există operații pe care funcțiile nu le pot îndeplini.

De exemplu:

```
#define MALLOC( n, type )  \  
    ( (type *) malloc ( ( n ) * sizeof ( type ) ) )
```

//apel:

```
pi = MALLOC( 25, int );
```

Macro versus functii

Dezavantaj macro-uri: efecte secundare

```
#define MAX( a , b )  ( ( a ) > ( b ) ) ? ( a ) : ( b )
```

...

```
x = 5; y = 8; z = MAX( x++ , y++ ) ;
```

```
printf("x = %d, y = %d, z = %d ",x , y , z );
```

//sau

```
z=((x++) > (y++))?(x++):(y++);
```

```
printf("x = %d, y = %d, z = %d ",x , y , z );
```

Afisare: x = 6, y = 10, z = 9

Macro versus funcții

Proprietate	Macro	Funcție
Dimensiune cod	Codul macro-ului este introdus în program la fiecare apel (program în creștere)	Codul funcției apare o singură dată
Viteza execuție	Foarte rapid	Timp adițional dat de apel/return
Evaluare argumente	Argumente evaluate cu fiecare folosire în cadrul macro-ului; pot apărea efecte secundare	Argumente evaluate o singură dată (înainte de apel); nu apar efecte secundare datorate evaluărilor multiple
Tip argumente	Macro-urile nu au tip; funcționează cu orice tip de argument compatibile cu operațiile efectuate	Argumentele au tipuri: sunt necesare funcții diferite pentru tipuri diferite de argumente, chiar dacă funcțiile execută același task

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

□ directiva **#ifdef**:

#ifdef nume	#ifdef nume
text	text1
#endif	#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

□ Exemplu:

```
#include <stdio.h>

#define medie(a,b,c) 0.4 * (a + b) + 0.2 * c

int main()
{
    int x = 2, y = 3, z = 4;

    #ifdef medie
    printf("medie = %f ", medie(x , y , z));
    #else
    printf("medie extra = %f ", 0.4 * (x + y) + 0.2 * z );
    #endif

    return 0;
}
```

Compilarea condiționată

□ directiva **#ifndef**:

#ifndef nume	#ifndef nume
text	text1
#endif	#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

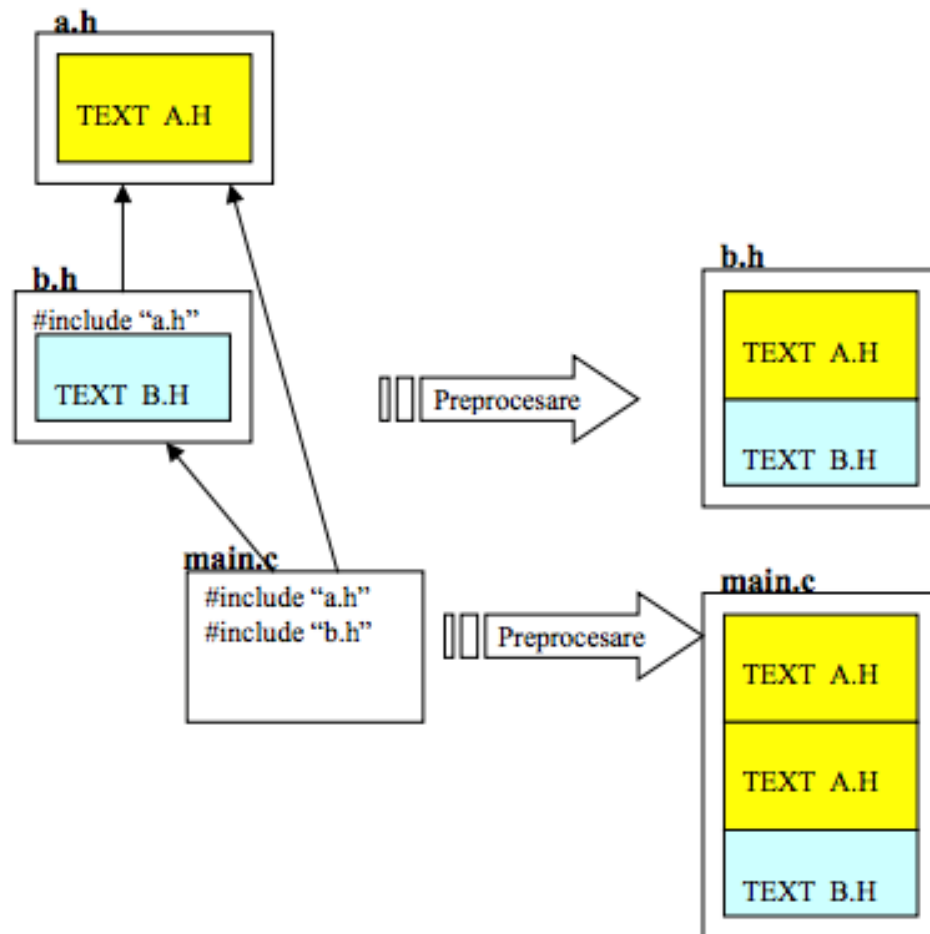
- ❑ directivele **#ifdef** și **#ifndef** sunt folosite de obicei pentru a evita incluziunea multiplă a modulelor în programarea modulară

- ❑ 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_H_  
#define      _MODUL_H_  
...  
#endif /*    _MODUL_H_    */
```


Macro-uri predefinite

- 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

Macro-uri predefinite

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

int main()
{
#ifdef 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;
#ifdef 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 4

1. **Tipuri derivate de date: structuri, uniuni, câmpuri de biți, enumerări**
2. **Instrucțiuni de control**
3. **Directive de preprocesare. Macrodefiniții**

Cursul 5

1. Funcții de citire/scriere.
2. Etapele realizării unui program C.
3. Fișiere text
4. Funcții