

# Programarea Calculatoarelor

# Cursul 5: Preprocesarea în C. Programarea modulară. Tipuri de variabile

Ion Giosan

Universitatea Tehnică din Cluj-Napoca Departamentul Calculatoare



## Preprocesarea în C

- Preprocesarea apare înaintea procesului de compilare
- Constă în substituirea simbolurilor din codul sursă pe baza directivelor de preprocesare
- Directivele de preprocesare sunt precedate de caracterul diez #
- Preprocesarea asigură
  - Includerea conținutului fișierelor (de obicei a fișierelor *header*)
  - Definirea de macrouri
  - Compilarea condiţionată



## Includerea fișierelor *header*

- Directiva de includere este specificată astfel:
  - #include <fisier header> dacă este o bibliotecă standard
  - #include "fisier\_header" dacă este o bibliotecă utilizator
- Înainte de compilare directiva este înlocuită cu conținutul fișierului respectiv
- Observaţii
  - Un fișier inclus poate conține alte directive #include
  - Directivele #include se plasează la începutul fișierului pentru ca vizibilitatea conținutului inclus (a definițiilor) să fie în întregul fișier
  - Directivele #include sunt utilizate frecvent pentru programe de dimensiuni mari
- Exemple

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



## Constante simbolice și macro-uri

- Directiva utilizată este #define
- Definirea unei constante simbolice este un caz special al definirii unui macro

#### #define nume text

- În timpul preprocesării nume este înlocuit cu text
- text poate să fie mai lungă decât o linie, continuarea se poate face prin caracterul \ pus la sfârșitul liniei
- text poate să lipsească, caz în care se definește o constantă vidă
- Înlocuirea se continuă până în momentul în care nume nu mai este definit sau până la sfârșitul fișierului
  - Renunțarea la definirea unei constante simbolice se poate face cu directiva #undef nume



# Constante simbolice și macro-uri

Definirea unui macro se aseamănă cu definirea unei funcții

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

- Numele macro-ului este nume
- Parametrii macro-ului sunt p1, p2, ..., pn
- Textul substituit este text
- Parametrii formali sunt substituiți de cei actuali în text
- Dacă textul se întinde pe mai multe linii se folosește același caracter \ pentru continuarea pe mai multe linii
- Apelul macro-ului este similar apelului unei funcții

```
nume(p_actual1, p_actual2, ..., p_actualn)
```



# Constante simbolice și macro-uri

```
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;
```



## Funcții vs. Macro-uri

- Invocarea unei funcții presupune apelul și execuția instrucțiunilor din corpul funcției
  - La apel tipul parametrilor este luat în considerare
- 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 pozițională
- Timpul de procesare este mai scurt când se utilizează macro-uri (apelul funcției necesită timp suplimentar)
- În C++ există funcții inline care sunt asemănătoare macro-urilor dar care verifică totodată și tipul parametrilor



- Facilitează dezvoltarea dar în special testarea codului
- Directivele care pot fi utilizate #if, #ifdef, #ifndef
- #if:

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

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



#### #ifdef:

- 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 dupa #endif



#### #ifndef:

- 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 dupa #endif



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

```
#ifndef _MODUL_H_
#define _MODUL_H_
... // continutul fisierului header
#endif /* _MODUL_H_ */
```

• Există o serie de nume predefinite care nu trebuie re/definite. Ex:

```
data compilării

CDECL apelul funcției urmărește convențiile C

STDC definit dacă trebuie respectate strict regulile ANSI C

numele complet al fișierului curent compilat

FUNCTION numele funcției curente

numărul liniei curente
```



 Directiva #error cauzează o eroare de compilare cu textul specificat ca și parametru

Declararea constantelor

```
tip const identificator=valoare;
sau
    const tip identificator=valoare;

Exemple:
    int const alpha=10;
    const double beta=20.5;
```

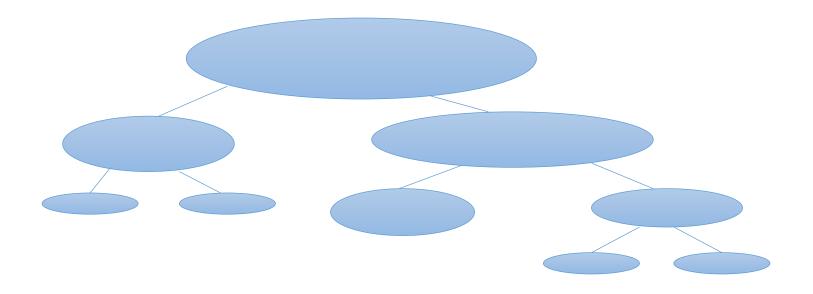


```
#include <stdio.h>
//constante simbolice
#define DEBUG
\#define X -3
#define Y 5
int main()
#ifdef DEBUG
   printf("Suntem in functia %s\n", FUNCTION ); //main
#endif
#if X+Y
   double a=3.1:
#else
   double a=5.7;
#endif
    a*=2;
#ifdef DEBUG
   printf("La linia %d valoarea lui a este %f\n", LINE ,a); //18 6.2
#endif
    a+=10;
   printf("a este %f",a); //16.2
    return 0;
```



## Programarea modulară

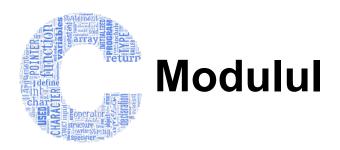
- Ajută la rezolvarea problemelor complexe care necesită un volum mare de cod scris pentru rezolvarea lor
  - Divizarea problemei în sub-probleme și rezolvarea acestora
  - Combinarea rezultatelor sub-problemelor rezolvate





## Avantajele programării modulare

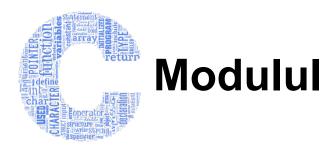
- Modulele pot fi scrise şi testate separat
- Modulele pot fi reutilizate
- Proiectele de dimensiuni mari pot fi dezvoltate în paralel de mai mulți programatori – lucrul în echipă
- Reduce dimensiunea programului
- Face programul mult mai ușor de citit
- Promovează conceptul de abstractizare
  - Fiecare modul ascunde detaliile despre task-urile implementate
  - Trebuie să se ştie ce fac task-urile fiecărui modul
  - Nu trebuie să se ştie cum sunt implementate task-urile fiecărui modul



 O colecție de funcții care rezolvă task-uri înrudite pentru rezolvarea unei probleme

#### Exemple:

- Modul care implementează funcții pentru lucrul cu numere complexe
- Modul care implementează funcții pentru lucrul cu matrice
- Modul care implementează funcții pentru lucrul cu informații din fișiere
- etc...



- Este împărțit în două părți
  - Partea publică
  - Partea privată

#### Partea publică

- Spune utilizatorului cum să apeleze funcțiile conținute de modulul respectiv
- Conține definiții de structuri de date și funcții care sunt utilizate în afara modulului
  - Aceste definiții sunt puse într-un fișier header (cu extensia .h)
  - Fişierul *header* trebuie inclus în fiecare program care depinde de modulul respectiv (folosește funcții sau structuri de date definite în modul)



#### Modulul

- Partea privată
  - Tot ce se află în interiorul unui modul este privat
  - Tot ce nu trebuie folosit direct din exteriorul unui modul trebuie să fie păstrat privat
- Exemplu de modul şi utilizarea lui într-un program

Definiții importate prin directiva
#include "complex.h"

Program care folosește
numere complexe și
operații cu numere complexe

complex.h

**Definirea** numerelor complexe și a operațiilor posibile (PUBLIC)

complex.c

Implementarea operațiilor posibile pe numere complexe (PRIVAT)



## Programarea modulară în C

- Presupune divizarea codului scris
  - Într-un fișier header (ex. complex.h) și
  - Fișierul sursă C corespunzător lui (ex. complex.c)
- Fişierul header va conţine doar ceea ce este permis a fi vizibil şi utilizabil din programul care foloseşte modulul:
  - Declarații de constante, tipuri, variabile globale
  - Prototipuri de funcții
- Orice altceva ce trebuie să rămână privat modulului trebuie să fie conținut în fișierul sursă (cu extensia .c)
- Fișierul header al modulului va conține declarațiile între directive de preprocesare specifice
  - Previne ca aceleași declarații să fie încărcate de mai multe ori în procesul de compilare



## Schelet de fișier header

```
/*
module1.h - Exemplu de fisier header pentru modulul "module1"
Informatii generale despre modul
*/
#ifndef MODULE1 H
#define MODULE1 H
/* Includerea bibliotecilor necesare pentru declaratiile ce
  urmeaza */
#include <stdlib.h>
#include <math.h>
#include "altFisierHeader.h"
/* Declaratii de constante */
/* Declaratii de tipuri */
/* Declaratii de variabile globale */
/* Prototipuri de functii */
#endif
```



## Programarea modulară în C

- Fișierul sursă C al modulului
  - Trebuie să includă mai întâi fișierele header necesare
  - Trebuie să includă apoi propriul fișier header
- Prin includerea propriului fișier header
  - În fișierul cu cod al modulului sunt alocate și inițializate variabilele globale declarate în *header*
  - Prototipurile funcțiilor sunt verificate cu antetul funcțiilor actuale care sunt implementate
    - Permite compilatorului să genereze mesaje de eroare în cazul în care acestea nu coincid (ex. din cauza numărului de parametri diferit, din cauza tipului diferit al parametrilor)
- Macro-urile, constantele şi tipurile de date declarate în interiorul unui fişier sursă (.c) nu pot fi exportate, ele sunt întotdeauna private



# Schelet de fișier sursă C

```
/* module1.c - Fisierul sursa corespunzator header-ului module1.h */
#include <stdio.h>
#include <string.h>

/* Includerea propriului fisier header! */
#include "module1.h"

/* Macro-uri si constante private */
/* Tipuri private */
/* Variabile globale private */
/* Functii private */
/* Functii publice */
```



#### Tipuri de variabile

- Variabile globale
  - Definite la începutul unui fișier sursă
  - Vizibile din punctul definirii lor până la sfârșitul fișierului sursă respectiv

```
tip identificator {, identificator};
static tip identificator {, identificator};
```

- Trebuie utilizate cu atenție deoarece:
  - Introduc dependențe între diferitele părți ale aceluiași program
  - Fac programul mai greu de citit
  - Fac programul mai greu de întreţinut
  - Pot să genereze coliziuni de nume
- Sunt inițializate automat
  - Numerele cu 0
  - Tablourile cu numere cu elemente de 0
  - Pointerii cu adresa NULL (0)



## Tipuri de variabile

#### Variabile externe

- Vizibile din alte fișiere sursă altele decât cel care conține definiția lor
- Acolo unde se dorește să fie vizibile se specifică cu ajutorul extern extern tip identificator {, identificator};
- Pot fi declarate
  - După antetul unei funcții vizibilitate doar în interiorul funcției
  - La începutul unui fișier sursă vizibilitate în toate funcțiile din acel fișier sursă

#### Variabile locale

- Se declară în interiorul unei funcții sau în interiorul unui bloc de instrucțiuni
- Vizibile doar în interiorul acelei funcții sau respectiv bloc de instrucțiuni
- Sunt neinițializate după declarație (au o valoare nedeterministă)



#### Tipuri de variabile

#### Tipuri de variabile locale

- Automate (de stivă)
  - Alocate pe stivă în timpul execuției
  - Trăiesc până la ieșirea din funcție sau respectiv până la părăsirea din blocul de instrucțiuni
  - Sunt re-create ori de câte ori se reintră în acea funcție/bloc

```
int a, b, c;
```

#### Statice

- · Alocate de compilator într-o zonă specială
- Persistă de-a lungul execuției programului
- Nu pot fi declarate externe în alt modul (sunt private modulului)

```
static int x, y, z;
```

#### Registru

Alocate în regiștrii procesorului

```
register float f;
```

 Compilatoarele moderne nu au nevoie de asemenea declarații – ele optimizează codul mai bine decât am putea face noi prin declararea de variabile register!



#### Ascunderea variabilelor

- Are loc atunci când apare o variabilă declarată într-un anumit bloc curent și care are un nume identic cu numele altei variabile declarată într-un bloc exterior
- Din acest moment, în blocul curent
  - Prin specificarea numelui variabilei se accesează conţinutul variabilei declarată local blocului
  - Este imposibil accesul la conţinutul variabilei declarată în blocul exterior (variabila locală blocului curent o ascunde pe cea din blocul exterior)
- La ieşirea din blocul curent se poate re-accesa variabila din blocul exterior
- Observaţie
  - Procedeul de ascundere poate fi apărea recurent în cazul blocurilor imbricate: o variabilă poate avea același nume în mod repetat în mai multe blocuri exterioare blocului curent



## Exemplu: ascunderea variabilelor

```
#include <stdio.h>
int a = 10; //variabila "a" globala
void f(){
   printf("2: a=%d\n", a); //variabila globala
    int a = 20; //variabila "a" locala functiei
   printf("3: a=%d\n", a); //variabila locala functiei
    if (a>0) //variabila locala functiei
        printf("4: a=%d\n", a); //variabila locala functiei
        int a = 30; //variabila "a" locala blocului curent
        printf("5: a=%d\n", a); //variabila locala blocului curent
        a = 40; //variabila locala blocului curent
   printf("6: a=%d\n", a); //variabila locala functiei
    a = 50; //variabila locala functiei
                                                    Rezultate afisate:
                                                    1: a=10
                                                    2: a=10
int main() {
   printf("1: a=%d\n", a); //variabila globala
                                                    3: a=20
    f():
                                                    4: a=20
   printf("7: a=%d\n", a); //variabila qlobala
                                                    5: a=30
    a = 60; //variabila globala
                                                    6: a=20
   printf("8: a=%d\n", a); //variabila globala
                                                    7: a=10
    return 0:
                                                    8: a=60
```



#### Cuvântul rezervat static

- Scris înaintea antetului unei funcții specifică faptul că acea funcție nu poate fi vizibilă din alt modul, fiind privată modulului respectiv
  - Toate funcțiile la care nu trebuie să se permită acces din exterior trebuie să fie declarate static
- Scris înaintea unei variabile globale specifică faptul că acea variabilă globală nu poate fi vizibilă din alt modul (declarată externă), fiind privată modulului respectiv
  - Toate variabilele globale la care nu trebuie să se permită acces din exterior trebuie să fie declarate static
- Scris înaintea unei variabile locale specifică faptul că acea variabilă este asemenea unei variabile globale doar pentru funcția respectivă
  - Acea variabilă poate fi inițializată în momentul declarării, linia de cod respectivă executându-se o singură dată la primul apel al funcției



# Exemplu: module și variabile

intregi.h

intregi.c

```
#ifndef INTREGI_H_INCLUDED
#define INTREGI_H_INCLUDED
int prim(int x);
#endif
//INTREGI H INCLUDED
```

```
#include <math.h>
#include <stdio.h>
#include "intregi.h"
//functie care nu poate fi vizibila din
exteriorul modulului
static int divizibil(int x, int d) {
    return (x%d==0);
int prim(int x) {
    static int nr apeluri=0; //variabila
                            locala statica
    nr apeluri++;
    printf("Apelul numarul %d al functiei
            prim pentru numarul %d!\n",
            nr apeluri,x);
    if (x<2) return 0;
    int limita; //variabila locala
                 automata, neinitializata
    limita=sqrt(x)+0.001;
    for (int i=2; i<=limita; i++)</pre>
        if (divizibil(x,i)) return 0;
    return 1;
```



# Exemplu: module și variabile

sir intregi.h

```
#ifndef SIR_INTREGI_H_INCLUDED
#define SIR_INTREGI_H_INCLUDED

#define CAPACITATE 100
void adauga_prim_in_sir(int x);
void afiseaza_sir();

#endif
// SIR_INTREGI_H_INCLUDED
```

#### sir\_intregi.c

```
#include <stdio.h>
#include "intregi.h"
#include "sir intregi.h"
static int v[CAPACITATE]; //variabila
globala statica, sir initializat cu
'CAPACITATE' elemente de 0, nu poate
fi accesat din alte module
static int nr elemente; //variabila
globala statica, initializata cu 0, nu
poate fi accesata din alte module
int nr; //variabila globala,
initializata cu 0, poate fi accesata
din alte module
void adauga_prim in sir(int x) {
    if (prim(x)) {
        v[nr elemente++]=x;
        nr=nr elemente;
void afiseaza sir() {
    printf("Sirul este: ");
    for (int i=0; i<nr elemente; i++)</pre>
        printf("%d ",v[i]);
    printf("\n");
```



# Exemplu: module și variabile

#### main.c

```
#include <stdio.h>
#include "intregi.h"
#include "sir intregi.h"
int main()
    // divizibil(10,2); -- eroare de
compilare, functia divizibil nu este poate
fi accesata, fiind statica
    /* extern int nr elemente;
       nr elemente=15; -- eroare de
compilare, nu se poate accesa variabila,
aceasta fiind statica
    */
    prim(3);
    adauga prim in sir(40);
    adauga prim in sir(43);
    adauga prim in sir(19);
    extern int nr;
    printf("Exista %d numere in sir!\n",nr);
    nr=15;
    adauga prim in sir(2);
    printf("Exista %d numere in sir!\n",nr);
    afiseaza sir();
    return 0;
```

#### Rezultate afișate

```
Apelul numarul 1 al functiei prim pentru numarul 3 !

Apelul numarul 2 al functiei prim pentru numarul 40 !

Apelul numarul 3 al functiei prim pentru numarul 43 !

Apelul numarul 4 al functiei prim pentru numarul 19 !

Exista 2 numere in sir!

Apelul numarul 5 al functiei prim pentru numarul 2 !

Exista 3 numere in sir!

Sirul este: 43 19 2
```