Programarea Calculatoarelor
Cursul 5

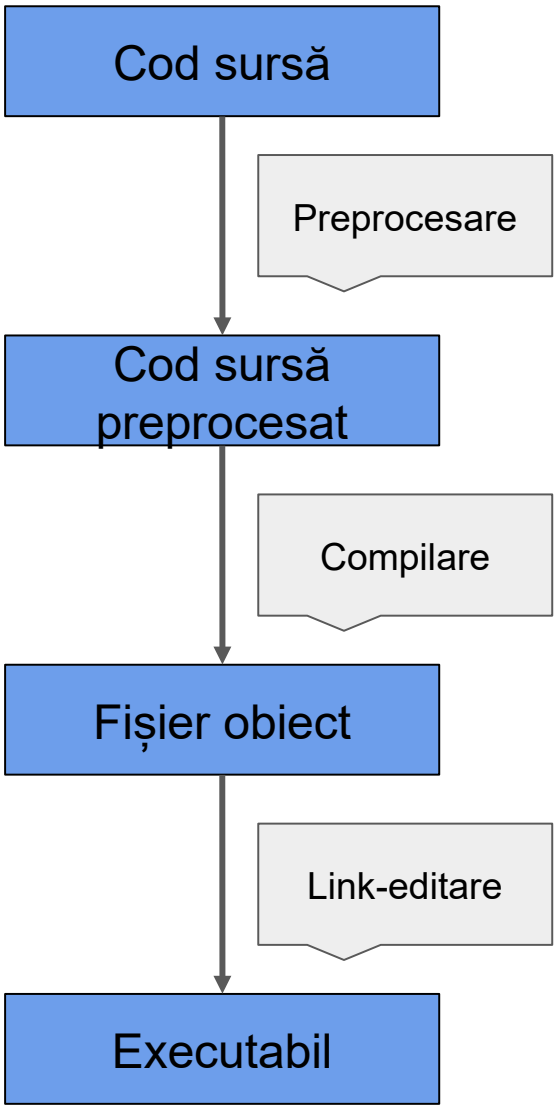
Directive de preprocesare
Programare modulară

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Obiective

- Cum se transformă codul sursă în executabil?
- Ce fac exact instrucțiunile care încep cu #?
- Cum scriem programe compuse din mai multe fișiere sursă?

Etapele creare executabil



- Codul sursă este transformat în mai multe etape
- Prima dată se efectuează operații de procesare de text
  - copiere, înlocuire text
- Apoi se compilează sursa
  - se verifică corectitudinea sintaxei
  - se generează fișierul obiect
- Apoi se leagă împreună mai multe fișiere obiect
  - main și biblioteci
  - se traduce în cod mașină
  - se generează executabilul

Directive de preprocesare

- Sunt comenzi pentru preprocesor
- Fiecare începe cu simbolul #
- Reprezintă operații de procesare de text care se efectuează înaintea compilării codului sursă

```
#include

#define #undef

#if #ifdef #ifndef #else #elif #endif
```

Includerea fișierelor

- Se face cu directiva `#include`
- Se copiază întregul conținut al fișierului inclus în fișierul de unde se face includerea
- Cel mai des se folosește pentru includerea fișierelor de tip antet sau header (eng.)
- Un fișier header conține
  - constante, definire tipuri, structuri, enumerări
  - prototipuri de funcții - fără implementare (fără definiție)
- Includerea antetului permite folosirea funcțiilor cu prototipul din header dacă este inclusă definiția lor în procesul de link-editare
  - se folosește <> pentru biblioteci și "" pentru header local

```
#include <header_biblioteca.h>
#include "header_utilizator.h"
```

Includerea fișierelor - exemplu

```c
#include <stdio.h>

int main(){
    return 0;
}
```

- Se poate genera fișierul preprocesat folosind comanda

```
gcc -E main.c > main.pre
```

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "c:/Program Files/mingw-w64/x86_64-8.1.0-posix-seh-rt_v6-rev0/mingw64/x86_64-w64-mingw32/include/stdio.h" 1 3
…

int __attribute__((__cdecl__)) printf(const char * __restrict__ _Format,...);
…

# 3 "main.c"
int main(){
        return 0;
}
```

Constante simbolice

- Se utilizează directiva `#define`
- Se înlocuiesc în fișierul sursă toate aparițiile ale primului string cu cel de-al doilea string - string replace eng.

```
#define to_replace replace_with
```

- Dacă al doilea șir este mai lung decât un rând atunci fiecare rând se termină cu caracterul \ (backslash)
- Înlocuirea se termină dacă se întâlnește perechea care specifică terminarea definiției sau la finalul fișierului

```
#undef to_replace
```

Constante simbolice predefinite

__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

Constante simbolice - exemplu

```c
#include <stdio.h>

//constante simbolice
#define ALPHA 30
#define BETA ALPHA+10

int main(){
        printf("%d\n", ALPHA);
        printf("%d\n", BETA*BETA);
        printf("%d\n", __LINE__);
        return 0;
}
```

```c
...
int main(){
        printf("%d\n", 30);
        printf("%d\n", 30 +10*30 +10);
        printf("%d\n", 11);
        return 0;
}
```

- Se poate genera fișierul preprocesat folosind comanda

```
gcc -E main.c > main.pre
```

Alternativă la constante simbolice

- Declararea constantelor
  - Alternativă programatică mai sigură la constante simbolice
  - Similară cu declararea variabilelor
  - Se adaugă cuvântul cheie `const` și se inițializează pe loc

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

sau

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

- Exemple:

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

- Greșit:

```
const int gamma;
gamma = 2;
```

Macro-uri

- Se realizează tot folosind `#define`
- Sunt asemănătoare unor funcții
  - Se recomandă folosirea funcțiilor în loc de macro-uri
  - Macro-uri putem folosi cu grijă pentru funcții scurte, de ex. min, max, interschimbare, valoare absolută

```
#define macro(p1, p2, …, pn) corp
```

- `macro` este numele, `p1, p2`, … sunt parametrii
- Când apare `macro` în text este înlocuit cu `corp`, care poate conține formule bazate pe parametrii macroului
- Apelul se face la fel ca la o funcție

```
macro(p1, p2, ... , pn)
```

Macro-uri - exemplu

```c
#include <stdio.h>
#define MIN(a,b) (((a)<(b))?(a):(b))
#define ABS1(x) (x<0)?-x:x
#define ABS2(x) (((x)<0)?-(x):(x))
#define INTER(tip,a,b) \
{tip c; c=a; a=b; b=c;}

int main(){
    int a = 1, b = 2;
    int c = MIN(a, b);
    int d = -ABS1(4-2);
    int e = -ABS2(4-2);
    INTER(int, a, b);
    printf("%d %d %d %d %d",
        a, b, c, d, e);
    return 0;
}
```

```c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"

int main(){
  int a = 1, b = 2;
  int c = (((a)<(b))?(a):(b));
  int d = -(4-2<0)?-4-2:4-2;
  int e = -(((4-2)<0)?-(4-2):(4-2));
  {int c; c=a; a=b; b=c;};
  printf("%d %d %d %d %d",
         a, b, c, d, e);
  return 0;
}
```

- Se poate genera fișierul preprocesat folosind comanda:

```
gcc -E main.c > main.pre
```

Compilarea condiționată

- Este posibilă prin folosirea lui `#if` și directive înrudite
- Dacă expresia după `#if` este adevărată se include partea din fișier până la următorul `#endif`, `#else` sau `#elif`
- Nu controlează fluxul execuției
- Util dacă vrem să dezactivăm părți din cod

```
#if expr
   text
#else
   tex2
#endif
```

```
#if expr
   text1
#endif  //sau #elif expr2
```

Compilarea condiționată

- Poate fi condiționată de existență sau inexistența unei constante simbolice definite în prealabil cu `#define`
- În acest caz se folosește `#ifdef` respectiv `#ifndef`
- Șfârșitul blocului se delimitează cu `#endif`
- Folosit în fișiere header pentru a împiedica includerea lor multiplă - header guard eng.

```
#ifdef expr
   text
#endif
```

```
#ifndef expr
   text1
#endif
```

Header guard

header.h

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int magic = 42;

#endif
```

- dacă nu există definită constanta simbolică HEADERFILE_H
- atunci se definește acum
- tot ce apare până la endif este inclus

main.c

```
#include <stdio.h>
#include "header.h"
#include "header.h"
int main(){
        printf("%d", magic);
        return 0;
}
```

- la biblioteci folosim <...> la antete locale "..."
- se include prima data fișierul - se definește constanta simbolică și se copiază conținutul fișierului
- se include a doua oară fișierul - fiindcă deja este definită constanta simbolică restul fișierului nu este inclus
- împiedicăm includerea dublă care se întâmplă frecvent la programe cu multe fișiere sursă și multiple include-uri

Programare modulară

- Presupune separarea programului în module independente
- Fiecare modul este responsabil de sarcini specifice
- De obicei fiecare modul este scris în alt fișier sursă
- Se face diferența dintre
  - interfața publică - Application Programming Interface API eng.
    - arată cum comunică modulul cu exteriorul
    - constante publice, definiție de tipuri, prototipuri de funcții
  - implementare privată
    - partea de cod pentru rezolvarea problemei
- Implementarea este privată pentru a asigura independența modulelor

Programare modulară în C

- Fiecare modul este scris în două fișiere dedicate
  - fișier header (antet) - conține constante publice, prototipuri (altele)
  - fișier sursă - conține implementarea privată (definiția funcțiilor)
- Un modul care dorește să folosească un alt modul
  - să includă fișierul header
  - să link-editeze fișierul obiect pentru acel modul la executabil
- Fișierul sursă
  - să includă bibliotecile necesare și fișierul header propriu
- Important
  - nu se include codul sursă
  - în acest fel nu se repetă codul
  - un modul scris poate fi folosit în mai multe proiecte
  - implementarea rămâne privată

Programare modulară în C - exemplu

conversii.h

```
#ifndef CONVERSII_H
#define CONVERSII_H

void showbinary(long long nr);
int to_digits(long long nr, int digits[], int b);
long long from_digits(int digits[], int n, int b);

#endif
```

conversii.c

```
#include <stdio.h>
#include "conversii.h"
void showbinary(long long nr){
        if (nr > 1)
                showbinary(nr/2);
        printf("%d", nr&1);
}
```

main.c

```
#include <stdio.h>
#include "conversii.h"
int main(){
        showbinary(23);
        printf("\n");
        return 0;
}
```

- funcțiile nefolosite pot rămâne neimplementate

Tipuri de variabile

- Variabile globale
  - Definite în afara funcțiilor
  - Vizibile din punctul definirii lor până la sfârșitul fișierului sursă
- Trebuie utilizate cu atenție deoarece:
  - Introduc dependențe între diferitele părți ale aceluiași program
  - Apar bug-uri prin modificarea lor neintenționată
  - Fac programul mai greu de citit
  - Fac programul mai greu de întreținut
  - 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;
```

  - Pot fi declarate
    - În blocul 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 acelui 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 recreate ori de câte ori se reintră în acea funcție/bloc
  - Statice
    - Nu sunt alocate pe stivă ci în zona statică
    - Persistă de-a lungul execuției programului
    - Nu pot fi declarate externe în alt modul (sunt private modulului)

```
static tip identificator;
```

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 dacă acolo se declarată ca externă, fiind **privată** primului modulului
  - 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

Cuvântul rezervat static - exemplu

static.c

```c
#include <stdio.h>
#include "static.h"

static const double PI = 3;
const double E = 2;

static void private_f(){
        puts("in private f");
}

void f(){
        private_f();
}
void add(int dx){
        static int s = 4;
        s += dx;
        printf("s = %d\n",
s);
}
```

static.h

```c
void f();
void add(int dx);
```

main.c

```c
#include <stdio.h>
#include "static.h"
extern const double PI; //PI nu se poate folosi
extern const double E;

int main(){
        //private_f(); //inaccesibil
        f();
        add(5);
        add(6);
        printf("%f\n", E);
        return 0;
}
```