

### Domeniul de vizibilitate al identificatorilor

Pt. orice identificator, compilatorul trebuie să-i decidă semnificația  
*Identificatorii obișnuiți*: variabile, tipuri, funcții, constante enumerare  
au un *spațiu de nume* comun (NU: variabilă și funcție cu același nume)

Q1: Un identificator poate fi folosit într-un punct de program ?

R: *Domeniul de vizibilitate* (al unei declarații / al unui identificator)

– domeniu de vizibilitate la nivel de *fișier* (*file scope*)  
pentru identificatori declarați în afara oricărui bloc (oricărei funcții)  
din punctul de declarație până la sfârșitul fișierului compilat  
– domeniu de vizibilitate la nivel de *bloc* (*block scope*)  
pentru identificatori declarați într-un bloc { } (corp de funcție,  
instrucțiune compusă) și pentru parametrii unei funcții  
din punctul de declarație până la acolada } care închide blocul

Un identificator poate fi *redeclarat* într-un bloc interior și își recapătă  
vechea semnificație când blocul ia sfârșit.

## Declarații. Instrucțiuni

19 octombrie 2005

### Domeniu de vizibilitate: Exemplu

```
int m, n, p; float x, y, z; /* m1, n1, p1, x1, y1, z1 */
void f(int n, int x) { /* n2, x2: alt n, alt x */
    int i; float y = 1; /* i1, y2 */
    m = p; p = n; /* m1 = p1; p1 = n2; */
    for (i = 0; i < 10; ++i) {
        float x = i*i; /* x3 = i1 * i1; */
        z += x; /* z1 += x3; */
    }
    z += x + y; /* z1 += x2 + y2 */
}
void main(void) {
    int i=0, m=3, x=2; /* i2, m2, x4 */
    z = f(m, x); /* z1 = f(m2, x4); */
    x = f(i, y); /* x4 = f(i2, y1); */
}
```

### Legătura dintre identificatori (linkage)

Q2: Două declarații ale unui identificator se referă la aceeași entitate?

R: Tipul de legătură (*linkage*) al unui identificator (obiect/funcție)

- *extern*: toate declarațiile identificatorului din toate fișierele care compun un program se referă la același obiect sau funcție pentru declarațiile *la nivel de fișier* fără specificator de memorare sau declarația cu specificatorul *extern* a unui identificator care nu a fost deja declarat cu tipul de legătură *intern*
- *intern*: toate declarațiile identificatorului din fișierul curent se referă la același obiect sau funcție; nu se propagă în exteriorul fișierului pt. declarațiile *la nivel de fișier* cu specificatorul de memorare *static*
- *fără legături* (*no linkage*): fiecare declarație denotă o entitate unică pentru declarațiile *la nivel de bloc* fără specificatorul *extern*

### Declarații, definiții tentative și definiții externe

În general, un identificator poate fi *declarat* de mai multe ori,  
dar poate fi *definit* o singură dată.

Pentru funcții, *declarația* și *definiția* au forme diferite:

declarația conține doar antetul, definiția conține și corpul funcției.

Un identificator fără legături nu va fi declarat de > 1 ori într-un bloc.

O *declarație* a unui identificator pt. un obiect e o *definiție externă*

– dacă declarația la nivel de fișier, cu un inițializator

– dacă declarația e la nivel de bloc, fără legături (vezi mai sus)

O declarație de obiect la nivel de fișier, fără inițializare, și fără specificator de memorare, sau cu specificatorul *static* e o *definiție tentativă*.

Dacă un identificator are în fișier una sau mai multe definiții tentative,  
dar nici o definiție externă, se comportă ca și pentru o definiție externă la nivel de fișier, cu inițializator nul.

### Declarații/definiții, tipuri de linkage. Exemplu

```
int x; /* x1, def. tentativă, linkage extern */
extern int y, z; /* y1, z1, declaratie, linkage extern */
static int m; /* m1, def. tentativă, linkage intern */
int y; /* y1, def. tentativă, vezi y mai sus */
extern int m; /* decl. link. intern, vezi m mai sus */
void f(int x); /* declarație funcție, linkage extern */
static int g(double); /* declarație funcție, linkage intern */
int h(int); /* declarație funcție, linkage extern */
void f(int m); /* decl. link. extern, vezi f mai sus */
extern int h(int y); /* decl. link. extern, vezi h mai sus */
void f(int x) { z=x; } /* definiția funcției f; z1 = x2; */
void main(void) /* definiția funcției main */
{
    static int m; /* m2, definiție, no linkage */
    int x; /* x3, definiție, no linkage */
    { extern int x; } /* declarație, linkage extern, x1 sus */
} /* g, h, z au definiții în alt fișier */
```

## Durata de memorare a obiectelor

Q3: Ce timp de viață/durată de memorare are un obiect în program?

R: 3 feluri diferite: *static*, *automatic* și *alocat* (discutat ulterior)

Pe întreaga durată de viață, un obiect are o *adresă constantă* și își *păstrează ultima valoare* memorată.

Durată de memorare *statică*:

- pentru obiecte declarate cu tipul de legătură *extern* sau *intern*, sau declarate cu specificatorul de memorare *static*
- timp de viață: *întreaga execuție* a programului.
- obiectul e *inițializat o singură dată*, înainte de lansarea în execuție.

Durată de memorare *automată*: pentru obiecte fără legătură

- timp de viață: de la intrarea în blocul asociat până la încheierea sa
- la fiecare apel recursiv, se crează o nouă instanță a obiectului
- *valoarea inițială: nedeterminată*;
- o eventuală inițializare în declarație e repetată de câte ori e atinsă

## Specificatori de memorare. Definiții de tip

engl. *storage class specifier*; se indică cel mult unul pe declarație.

**extern, static**: discutați mai sus

- stabilesc atât proprietățile de *linkage* cât și durata de memorare
- auto**: implicit pentru variabile declarate la nivel de bloc
- register**: indicație către compilator de a optimiza pentru viteză accesul la o variabilă (alocând pentru ea un registru dacă e posibil)
- nu se poate folosi operatorul adresă pentru variabile **register**

**Definiții de tip:** `typedef declarație`

```
typedef unsigned long size_t;           typedef unsigned char byte;
– dacă în declarație, identificadorul ar fi o variabilă de un anumit tip, atunci typedef declarație definește identificadorul ca numele celui tip
```

```
Ex: în int mat3x5[3][5];               mat3x5 ar fi o matrice de 3x5 întregi.
typedef int mat3x5[3][5];              /* mat3x5 e tipul tablou de 3x5 int */
mat3x5 A, B;                           /* A, B sunt variabile tablou de 3x5 int */
```

## Calificatori de tip (type qualifiers)

**const**

- specifică interzicerea modificării prin program a valorii obiectului
- ex. pt. declarații de constante: `const int MAX = 100;`
- nu se permite folosirea de operatori de atribuire pt. obiecte **const**
- compilatorul e liber să le aloce în memorie read-only

**volatile**

- obiectul poate fi modificat în mod necunoscut implementării (ex. port hardware, întrerupere asincronă, program concurent)
- ⇒ indicație către compilator să citească valoarea curentă din memorie la fiecare folosire, fără optimizări (cf. **register**)
- combinat cu **const**: obiectul poate fi modificat doar extern:
 

```
extern const volatile int real_time_clock;
```

**restrict**

- stabilește că un obiect poate fi modificat doar prin pointerul indicat (calificator folosit doar cu pointeri, permite optimizări de compilare)

## Instrucțiunile limbajului C

Instrucțiunea **expresie** `expresieopt ;`

- orice expresie, evaluată pentru efectele ei laterale; în particular: expresii de *atribuire*: `x = y + 1; y *= 2; --z;`
- *apel de funcție* (ignorând valoarea returnată): `printf("salut!\n");`
- instrucțiunea *vidă*; (expresia lipsește)

Exemplu: ciclu cu corp vid `while (s[i++]);`

Obs: în C ; nu e separator, ci *face parte* din anumite instrucțiuni

Instrucțiunea **compusă** (bloc) `{ lista-declarații lista-instrucțiuni }`

- grupează declarațiile/instrucțiunile din listă sintactic într-o instrucțiune poate fi încuibată (conține alte blocuri); poate fi vidă `{ }`
- *lista-declarații* nu poate conține *definiții* de funcții

În C99 (și în C++) un bloc poate conține declarații și instrucțiuni în orice ordine.

## Instrucțiuni etichetate

*identificator* : *instrucțiune*

**case** *expresie-constantă-int* : *instrucțiune*

**default** : *instrucțiune*

Etichetele au *spațiu de nume* separat de cel al identificatorilor obișnuiți (putem avea variabile/funcții/etc. și etichete cu același nume)

*Domeniul de vizibilitate* al etichetei e corpul funcției în care se află (numele de etichete trebuie să fie unice în cadrul unei funcții)

Etichetele **case** și **default** pot apare doar în instrucțiunea **switch**.

Într-o instrucțiune **switch** poate exista cel mult o etichetă **default** iar constantele întregi din etichetele **case** vor fi distincte.

## Instrucțiuni de selecție

Instrucțiunea **if** `if ( expresie ) instrucțiune1`  
sau `if ( expresie ) instrucțiune1 else instrucțiune2`

- expresia trebuie să fie de tip scalar (întreg, real, enumerare)
- dacă expresia e nenulă se execută *instrucțiune1*, altfel *instrucțiune2*
- un **else** e asociat întotdeauna cu cel mai apropiat **if**

Instrucțiunea **switch** `switch ( expresie-intreagă ) instrucțiune`

- se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)
- dacă în corpul *instrucțiune* (compusă) există o etichetă **case** cu valoarea întreagă obținută, se sare la instrucțiunea respectivă
- dacă nu, și există o etichetă **default**, se sare la acea instrucțiune
- altfel nu se execută nimic (se trece la instrucțiunea următoare)
- pt. același cod la mai multe etichete: `case val1: case val2: șir-instr`
- Obs: Execuția nu se oprește la următorul **case** (e doar o etichetă); ieșirea din **switch**: doar cu instruct. **break** sau la sfârșitul corpului!
- ⇒ permite utilizarea de cod comun pe ramuri, dar cu mare atenție!

### Instructiunea switch: exemplu

```
char c; int a, b, r;
printf("Scrieti o operatie intre doi intregi: ");
if (scanf("%d %c %d", &a, &c, &b) == 3) { /* toate 3 corect */
    switch (c) {
        case '+': r = a + b; break; /* iese din corpul switch */
        case '-': r = a - b; break; /* idem */
        default: c = '\0'; break; /* fanion caracter eronat */
        case 'x': c = '*'; /* 'x' e tot inmultire, continua */
        case '*': r = a * b; break; /* ca si pt.*' apoi iese */
        case '/': r = a / b; /* la sfarsit nu trebuie break */
    }
    if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
    else printf("Operatie necunoscuta\n");
} else printf("Format eronat\n");
```

### Instructiunile de ciclare

**Ciclul cu test inițial:** `while ( expresie ) instructiune`  
**Ciclul cu test final:** `do instructiune while ( expresie );`  
 – execută *instructiunea* cat timp valoarea expresiei e nenulă (adeverată)  
 expresia poate avea orice tip scalar (intreg, real)  
 – diferă momentul de evaluare a expresiei (inainte/dupa fiecare iteratie)  
 Obs: În Pascal, din repeat ... until se iese pe condiție *true* (invers!)

**Instructiunea for**

```
for (exp-init ; exp-test ; exp-cont)
    instructiune
    exp-init;
    while (exp-test) {
        instructiune;
        exp-cont;
    }
```

e echivalentă\* cu:

\* excepție: instructiunea continue, vezi ulterior  
 – oricare din cele 3 expresii poate lipsi (dar cele două ; rămân)  
 – dacă *exp-test* lipsește, e tot timpul adeverată (ciclu infinit)

În C99 (și C++) in loc de *exp-init* poate aparea o **declarație** de variabile (eventual inițializate) cu domeniu de vizibilitate toata instructiunea.  
`for (int i = 0; i < 10; ++i) { /*corp*/ } // i nu e vizibil dupa`

### Instructiunea return

```
return expresie_opt ;
```

– încheie execuția funcției curente  
 – returnează valoarea expresiei date (dacă este prezentă)  
 Obs: într-o funcție care nu are tipul void, fiecare cale prin cod trebuie să returneze o valoare (compilatorul avertizează în caz contrar)

```
int pos(char s[], char c) /* prima pozitie a lui c in s */
{
    int i = 0;
    do
        if (s[i] == c) return i; /* returnează poziția găsită */
        while (s[i++]);
    return -1; /* -1 ca fanion, nu s-a găsit */
}
```

– funcția main returnează un cod (succes/eroare) sistemului de operare  
 – uzual, se declară ca int și returnează implicit 0 (succes)

### Instructiunea break

– produce ieșirea din corpul instructiunii `while`, `do`, `for` sau `switch`  
*imediat înconjurătoare*; execuția continuă cu instructiunea următoare  
 – mai convenabilă decât testarea unei variabile boolene la ciclul următor  
 – mai lizibil, dacă codul peste care se sare e complex

```
const int MAX = 20;
int i, t[MAX], v;
/* caută pe v în tabloul t */
for (i = MAX; --i >= 0; )
    if (t[i] == v) break;
if (i == -1) printf("nu s-a găsit\n");
else printf("găsit la poziția %d\n", i);
```

### Instructiunea continue

– produce trecerea la sfârșitul iterației într-un ciclu `while`, `do` sau `for`  
 începând cu testul pt. `while` și `do`, și cu *expr3* (actualizare) pt. `for`  
 (controlul trece la punctul din ciclu de după ultima instructiune)  
 – la fel, cod mai lizibil, dacă partea neexecutată din iterație e complexă

```
for (d = 2; ; d++) { /* descompune n > 1 în factori primi */
    if (n % d != 0) continue; /* nu se împarte, următorul! */
    exp = 0;
    do /* repetă de câte ori d e factor */
        exp++;
    while ((n /= d) % d == 0);
    printf ("%d^%d ", d, exp); /* scrie factorul curent */
    if (n == 1) break; /* am terminat */
}
```

### Instructiunea goto

Sintaxa: `goto eticheta ;`  
 Efectul: se sare la execuția instructiunii cu *eticheta* specificată  
 Obs: orice instructiune poate fi etichetată opțional *etichetă : instr*

– instructiunea `goto` nu corespunde principiilor programării structurate  
 – de evitat: duce ușor la programe dificil de înțeles și analizat  
 – orice program poate fi rescris fără folosirea lui `goto`  
 (eventual utilizând teste și/sau variabile boolene suplimentare)  
 – poate fi totuși utilă, ex. pentru ieșirea din mai multe cicluri încuibate

```
while (...) { /* scriem într-un fișier, linie cu linie */
    while (...) { /* prelucrăm cuvintele și spațiile din linie */
        if (eroare_la_scriere)
            goto eroare; /* abandonează ciclurile */
    }
}
```

**eroare:** /\* cod pt. tratarea erorii \*/