Declaraţii. Instrucţiuni

19 octombrie 2005

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.

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.

Declaraţii/definiţii, 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);                  /* declaraţie funcţie, linkage extern */
static int g(double);           /* declaraţie funcţie, linkage intern */
int h(int);                     /* declaraţie funcţie, 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; }          /* definiţia funcţiei f; z1 = x2; */
void main(void)                 /* definiţia funcţiei main */
{
  static int m;                 /* m2, definiţie, no linkage */
  int x;                        /* x3, definiţie, no linkage */
  { extern int x; }             /* declaraţie, linkage extern, x1 sus */
}                               /* g, h, z au definiţii în alt fişier */
```

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. Definiţii de tip

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

`extern`, `static`: discutaţi mai sus

– stabilesc atât proprietăţile de *linkage* cât și durata de memorare

`auto`: implicit pentru variabile declarate la nivel de bloc

`register`: indicaţie către compilator de a optimiza pentru viteză accesul la o variabila (alocând pentru ea un registru dacă e posibil)

– nu se poate folosi operatorul adresă pentru variabile `register`

Definiţii de tip: `typedef` *declaraţie*

```
typedef unsigned long size_t;          typedef unsigned char byte;
```

– dacă în *declaraţie*, identificatorul ar fi o *variabilă* de un anumit tip, atunci `typedef` *declaraţie* definește identificatorul ca *numele* acelui 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)

Instrucţiunile limbajului C

Instrucţiunea *expresie* *expresie*$_{opt}$;

– *orice* expresie, evaluată pentru efectele ei laterale; în particular:

expresii de *atribuire*: `x = y + 1; y *= 2; --z;`

apel de funcţie (ignorând valoarea returnată): `printf("salut!\n");`

instrucţiunea *vidă* ; (expresia lipseşte)

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

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

Instrucţiunea *compusă* (bloc) { *lista-declaraţii* *lista-instrucţiuni* }

grupează declaraţiile/instrucţiunile din listă sintactic într-o instrucţiune

poate fi încuibată (conţine alte blocuri); poate fi vidă { }

lista-declaraţii nu poate conţine *definiţii* de funcţii

În C99 (şi în C++) un bloc poate conţine declaraţii şi instrucţiuni în orice ordine.

Instrucţiuni etichetate

identificator `:` *instrucţiune*
`case` *expresie-constantă-int* `:` *instrucţiune*
`default :` *instrucţiune*

Etichetele au *spaţiu de nume* separat de cel al identificatorilor obişnuiţi (putem avea variabile/funcţii/etc. şi etichete cu acelaşi nume)

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

Etichetele `case` şi `default` pot apare doar în instrucţiunea `switch`. Într-o instrucţiune `switch` poate exista cel mult o etichetă `default` iar constantele întregi din etichetele `case` vor fi distincte.

Instrucţiuni de selecţie

Instrucţiunea `if` `if ( expresie ) instrucţiune1`
sau `if ( expresie ) instrucţiune1 else instrucţiune2`
– expresia trebuie să fie de tip scalar (întreg, real, enumerare)
– dacă expresia e nenulă se execută *instrucţiune1*, altfel *instrucţiune2*
– un `else` e asociat întotdeauna cu cel mai apropiat `if`

Instrucţiunea `switch` `switch ( expresie-intreagă ) instrucţiune`
– se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)
– dacă în corpul *instrucţiune* (compusă) există o etichetă `case`
cu valoarea întreagă obţinută, se sare la instrucţiunea respectivă
– dacă nu, şi există o etichetă `default`, se sare la acea instrucţiune
– altfel nu se execută nimic (se trece la instrucţiunea următoare)
– pt. acelaşi cod la mai multe etichete: `case` *val1* `:` `case` *val2* `:` *şir-instr*
Obs: Execuţia nu se opreşte la următorul `case` (e doar o etichetă);
ieşirea din `switch`: doar cu instruct. `break` sau la sfârşitul corpului!
⇒ permite utilizarea de cod comun pe ramuri, dar cu mare atenţie!

Instrucţiunea `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 înmulţire, continuă */
    case '*': r = a * b; break;  /* ca şi pt.'*' apoi iese */
    case '/': r = a / b;    /* la sfârşit nu trebuie break */
  }
  if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
  else printf("Operaţie necunoscută\n");
} else printf("Format eronat\n");
```

Instructiunile de ciclare

Ciclul cu test iniţial: `while (` *expresie* `)` *instrucţiune*

Ciclul cu test final: `do` *instrucţiune* `while (` *expresie* `);`

– execută *instrucţiunea* cat timp valoarea expresiei e nenulă (adevărată)

expresia poate avea orice tip scalar (intreg, real)

– diferă momentul de evaluare a expresiei (înainte/după fiecare iteraţie)

Obs: În Pascal, din `repeat ... until` se iese pe condiţie *true* (invers!)

Instrucţiunea `for`

`for` (*exp-init* ; *exp-test* ; *exp-cont*)
instrucţiune

e echivalentă* cu:

exp-init;
`while` (*exp-test*) `{`
instrucţiune;
exp-cont;
`}`

* excepţie: instrucţiunea `continue`, vezi ulterior

– oricare din cele 3 expresii poate lipsi (dar cele două `;` ramân)

– dacă *exp_test* lipseşte, e tot timpul adevărată (ciclu infinit)

În C99 (şi C++) in loc de *exp-init* poate aparea o *declaraţie* de variabile (eventual iniţializate) cu domeniu de vizibilitate toata instrucţiunea.

```
for (int i = 0; i < 10; ++i) { /*corp*/ } // i nu e vizibil dupa
```

Instrucţiunea `return`

`return` *expresie*$_{opt}$;

– încheie execuţia funcţiei curente

– returnează valoarea expresiei date (dacă este prezentă)

Obs: într-o funcţie 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 poziţia găsită */
  while (s[i++]);
  return -1;                    /* -1 ca fanion, nu s-a găsit */
}
```

– funcţia `main` returnează un cod (succes/eroare) sistemului de operare

– uzual, se declară ca `int` şi returnează implicit 0 (succes)

Instrucţiunea `break`

– produce ieşirea din corpul instrucţiunii `while, do, for` sau `switch` *imediat înconjurătoare*; execuţia continuă cu instrucţiunea 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 poziţia %d\n", i);
```

Instrucțiunea `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 instrucțiune)
– 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 */
}
```

Instrucţiunea goto

Sintaxa: `goto` *eticheta* ;
Efectul: se sare la execuţia instrucţiunii cu *eticheta* specificată
Obs: orice instrucţiune poate fi etichetată opţional *etichetă* : *instr*

– instrucţiunea `goto` nu corespunde principiilor programării structurate
– de evitat: duce uşor la programe dificil de înţeles şi analizat
– orice program poate fi rescris fără folosirea lui `goto`
(eventual utilizând teste şi/sau variabile boolene suplimentare)
– poate fi totuşi utilă, ex. pentru ieşirea din mai multe cicluri încuibate

```
while (...) { /* scriem într-un fişier, linie cu linie */
  while (...) { /* prelucrăm cuvintele şi spaţiile din linie */
    if (eroare_la_scriere)
      goto eroare; /* abandonează ciclurile */
  }
}
eroare: /* cod pt. tratarea erorii */
```