# Tipuri definite de utilizator

8 noiembrie 2004

# Generalități. Sintaxă

Un tip definește o *mulțime de valori* și *operațiile* posibile cu acestea.
În C se pot defini de utilizator tipuri *enumerare, structură și uniune*.

În aceste cazuri, *specificatorul* (numele) de tip e format din cuvântul cheie `enum`, `struct` și `union`, urmat de un *identificator*.

– în folosirea ulterioară: `enum` *culoare* și nu doar: *culoare*
– dar se poate defini cu `typedef` un nume de tip de sine stătător

Identificatorul (eticheta, *tag*) unui astfel de specificator de tip e într-un spațiu de nume separat de identificatorii obișnuiți sau etichetele de instrucțiuni (dar comun pentru cele tipurile `enum`, `struct`, `union`)

Identificatorul (*tag*) e opțional dacă tipul e folosit doar o singură dată pentru declararea unor variabile (tip anonim).

# Tipuri enumerare

Folosite pentru a da nume simbolice unui șir de valori numerice.

Sintaxa: `enum` *identificator*$_{opt}$ `{` *lista-constante* `}` *listadeclaratori*$_{opt}$ `;`

– constantele pot avea specificate valori (și o valoare se poate repeta)

```
enum luni_curs {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};
```

– implicit, șirul valorilor e crescător cu pasul 1, iar prima valoare e 0

– un nume de constantă nu poate fi folosit în mai multe enumerări

– tipurile enumerare sunt tipuri întregi

⇒ variabilele enumerare se pot folosi la fel cu variabilele întregi

– cod mai lizibil decât prin declararea separată de constante

```
enum {D, L, Ma, Mc, J, V, S} zi; /* tip anonim */
int nr_ore_lucru[7]; /* număr de ore pe zi */
for (zi = L; zi <= V; zi++) nr_ore_lucru[zi] = 8;
```

# Definirea tipurilor: observaţii practice

– definirea unor nume de tip (`typedef`) facilitează înţelegerea codului

– tablouri: folosiţi dimensiuni constante simbolice (nu direct numere)
(modificările ulterioare sunt necesare într-un singur punct în program)

```
#define LEN 20    /* LEN e substituit cu 20 de preprocesor */
int a[LEN], i;
for (i = 0; i < LEN; ++i) { /* ceva */ }
```

– concepeţi structuri de date uşor de modificat şi de extins

– anticipaţi limitările care pot deveni rapid problematice
  – adresarea segmentată pe 16 biţi în procesoarele Intel (depăsită)
  – utilizarea a doar două cifre pentru an (problema anului 2000)
  – mai comun: limite fixe (şi mici) pentru lungimi de nume, adrese, linii de text, dimensiuni de fişiere, durate de timp, etc.

⇒ definiţi pentru acestea cu `typedef` tipuri modificabile ulterior

⇒ folosiţi tipurile oferite de limbaj (ex. `size_t`)

# Structuri

Folosite pentru gruparea mai multor elemente de tipuri de date diferite
– exemplu clasic: înregistrare din bază de date despre persoane

```
struct student {
  char *nume, *prenume; /* mai flexibil; nu tablou de dim. fixă */
  char *adresa;
  char nr_tel[10];      /* sau long, suficient pentru 9 cifre */
  float medie_an[5];    /* mediile pe ani de studiu */
  float nota_dipl;      /* nota la examenul de diploma */
};
```

`struct` *identificator*$_{opt}$ `{` *lista-câmpuri* `}` *lista-declaratori*$_{opt}$ `;`
– elementele unei structuri se numesc *câmpuri* (engl. fields)
– pot fi de orice tip, dar nu de *același* tip structură (nu recursiv)
– structuri de tip diferit pot avea fără conflict nume de câmpuri identice
– structuri, tablouri, uniuni = tipuri *agregate* (complexe, nu simple)

# Utilizarea structurilor. Operatori

Accesul la câmpuri: se face cu sintaxa *nume_variabila*.*nume_câmp*
operatorul de *selecţie* . (considerat operator postfix)

```
struct student s;
s.nume = "Stefanovici";
strcpy(s.telefon, "256123456");
s.medie_an[2] = 9.35;
```

Iniţializarea structurilor: câmp cu câmp, între acolade, ca şi pentru alte agregate: `struct point { float x, y; } pct1 = { 2.5, 1.5 };`

*Literale compuse* ( *nume-tip* ) { *iniţializatori* }
– sunt obiecte fără nume, de tipul indicat; pot fi utilizate în program

```
void drawpoint(struct point p);
struct point p2;
p2 = (struct point) { -1, 2 };
drawpoint((struct point) { 1.5, 2.5 });
```

# Utilizarea structurilor (cont.)

Structurile *pot* fi atribuite în totalitatea lor.

```
struct point p1, p2;  p1 = p2;
```

Structurile *pot* fi transmise către / returnate de funcţii.
Pt. dimensiuni mari, se preferă transmiterea / returnarea de pointeri.

Structurile *nu pot* fi comparate cu operatori logici
⇒ trebuie comparate individual câmpurile lor, sau (din `string.h` :)

```
int memcmp(const void *s1, const void *s2, size_t n);
```

(returnează 0 la egalitate, sau diferenţa între primii doi octeţi neegali)

```
struct { /* ceva câmpuri */ } x, y;
if (memcmp(&x, &y, sizeof x)) { /* sunt diferite */ }
```

# Pointeri la structuri

Frecvent: accesul la câmpuri prin intermediul unui pointer la structură:

```
struct student *p; /* p = ...  */ (*p).nota_dipl = 9.50;
```

Operatorul `->` e echivalent cu indirectarea urmată de selecție:

`pointer->nume_câmp` e echivalent cu `(*pointer).nume_câmp`

Operatorii `.` și `->` au precedența cea mai ridicată, ca și `()` și `[]`

Atenție la ordinea de evaluare !

| | | |
|---|---|---|
| `p->x++` | înseamnă | `(p->x)++` |
| `++p->x` | înseamnă | `++(p->x)` |
| `*p->x` | înseamnă | `*(p->x)` |
| `*p->s++` | înseamnă | `*((p->s)++)` |

# Structuri și tablouri

În C, tipurile agregat pot fi combinate arbitrar (tablouri de structuri, structuri cu câmpuri de tip tablou, etc.)

Tipurile trebuie definite în așa fel încât să grupeze logic datele.

Ex.: dacă două tablouri au același domeniu pt. indici și datele de la același indice sunt folosite împreună, e preferabilă gruparea în structură:

```
char* nume_luna[12] = { "ianuarie", /* ... , */ "decembrie" };
char zile_luna[12] = { 31, 28, 31, 30, /* ... , */ 30, 31 };
/* e preferabilă varianta următoare */
typedef struct {
  char *nume;
  int zile;
} tip_luna;
tip_luna luni[12] = {{"ianuarie",31}, /*...,*/ {"decembrie",31}};
```

# Structuri de date recursive

Un câmp al unei structuri nu poate fi o structură de același tip (s-ar obține o structură de dimensiune infinită/nedefinită!).
Poate fi însă *adresa* unei structuri de același tip (un pointer)!
⇒ structuri de date recursive, înlănțuite (liste, arbori, etc.)

```
struct wl {         /* tag-ul wl e necesar în declararea lui next */
  char *word;       /* informaţia propriu-zisă */
  struct wl *next; /* pointer la acelaşi tip de structură */
};                  /* defineşte tipul struct wl */
```

Un arbore binar, având în noduri numere întregi:

```
typedef struct t tree; /* defineşte tipul incomplet tree */
struct t {
  int val;
  tree *left, *right;  /* foloseşte numele din typedef */
};              /* tree şi struct t sunt complete şi echivalente */
```

# Pointeri pentru implementarea listelor

Folosirea *adreselor de adrese* poate conduce la cod mai uniform

```
typedef struct l { int key; struct l* next; } node_t;
typedef node_t *list_t; /* list_t e tipul adresă de node_t */

list_t *pos(list_t *p, int k) /* adresa unde s-ar afla k */
{
  while (*p && (*p)->key < k) p = &(*p)->next;
  return p; /* poziţia lui k sau unde ar trebui inserat */
}
int member(list_t l, int n) /* returnează boolean */
{
  list_t *p = pos(&l, n);
  return *p && (*p)->key == k;
}
```

# Pointeri și liste (cont.)

```
list findins(list_t *l, int n) /* caută şi inserează la nevoie */
{
  list_t new, *p = pos(l, n);
  if (*p && (*p)->key == k) return *p;
  if (!(new = malloc(sizeof node_t)) return NULL;
  new->key = k; new->next = *p; *p = new; return new;
}
list_t delete(list_t *l, int n) /* returnează nodul sau NULL */
{
  list pn, *p = pos(l, n);
  if (*p && (*p)->key == k) {
    pn = *p; *p = pn->next; return pn;
  } else return NULL;
}
```

# Câmpuri pe biţi

Se pot declara câmpuri întregi cu un număr specificat de biţi
⇒ Testarea/setarea unor biţi se face folosind direct numele câmpului
fără a fi nevoie de definirea de măști și utilizarea unor operatori pe biţi
*câmp* ::= *nume* : *int_const* ; | : *int_const* ;

```
struct packet {
  int : 2;          /* primii doi biţi nu interesează */
  int error: 1;     /* un bit, semnalizează eroare */
  int status: 3;    /* un câmp pe 3 biţi */
  int : 0;          /* forţează alinierea la octetul următor */
  int seq_no: 4;    /* număr de secvenţă pe 4 biţi */
} pkt;
if (pkt.error) { ... }
else if (pkt.status == 5) { ... }
else pkt.seq_no++;
```

# Uniuni

Agregate a căror valoare poate avea date de tipuri diferite, după caz.

Sintaxa: similară cu cea pentru structuri
`union` *opt_nume_tip* `{` *lista_câmpuri* `}` *opt_lista_declaratori* `;`

Lista de câmpuri este însă o listă de variante:
– o variabilă structură conţine *toate* câmpurile declarate
– o variabilă uniune conţine exact *una* din variantele date
(dimensiunea tipului e dată de cel mai mare câmp)

– o variabilă uniune nu conţine informaţii despre varianta reprezentată
– acest lucru trebuie memorat explicit în program (în altă variabilă)

# Utilizarea uniunilor

Exemplu: un analizor lexical (prima fază a compilatorului) returnează:
– un cod întreg pt. fiecare atom lexical (cuvânt cheie, operator, etc.)
– date suplimentare pentru identificatori (nume) și constante (valoare)

```
enum tok { IDENT, INUM, FNUM, DO, IF, ..., PLUS, ..., COMMA, ... };
typedef union {
  char *id;   /* şir de caractere pentru identificator */
  int ival;   /* valoare pentru constantă întreagă */
  float fval; /* valoare pentru constantă reală */
} lexvalue;
enum tok token;
lexvalue lv;
switch (token) {
  case IDENT: printf("%s", lv.id); break;
  case INUM: printf("%d", lv.ival); break;
  case FNUM: printf("%f", lv.fval); break;
}
```