Tipuri definite de utilizator

(enumerări, structuri, uniuni)

29 noiembrie 2005

Tipuri enumerare

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

Sintaxa: `enum` *identificator*$_{opt}$ { *lista-constante* } *listadeclaratori*$_{opt}$;
– constantele pot avea specificate valori (şi o valoare se poate repeta)
`enum luni_curs {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};`
– implicit, şirul 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; declara doar var.zi
// tipul nu are nume ==> nu mai putem declara altundeva variabile
int nr_ore_lucru[7]; /* număr de ore pe zi */
for (zi = L; zi <= V; ++zi) nr_ore_lucru[zi] = 8;
```


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; // lungimea poate varia dar trebuie alocat!
  char *adresa; // structura are loc pentru pointer, nu pentru sir!
  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 *acelaşi* tip structură (nu recursiv)
– structuri de tip diferit pot avea fără conflict nume de câmpuri identice
– structuri, tablouri, uniuni = tipuri *agregate* (complexe, nu simple)

Generalităţi. Sintaxă

Numele (*specificatorul*) de tip e format din cuvântul cheie `enum, struct` şi `union`, urmat de un *identificator*.

– folosit ca: `enum` *culoare*, `struct` *elev*; nu doar: *culoare*, *elev*
– dar se poate defini cu `typedef` un nume de tip de sine stătător

Eticheta (*tag*) unui astfel de specificator de tip e într-un spaţiu de nume *separat* de identificatorii obişnuiţi şi etichetele de instrucţiuni, dar *comun* pentru cele tipurile `enum, struct, union`
⇒ putem avea: `int l; struct l; goto l;` dar nu `enum l; struct l;`

Putem folosi diverse variante de sintaxa:

```
struct s { /* */ }; /* apoi declaram */ struct s x, y;
struct s { /* */ } x, y; /* declaram direct si var. */
struct { /* */ } x, y; // tip anonim, doar cu variabilele x, y
typedef struct s { /* */ } s_t; struct s x; s_t y; // la fel
```


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 });
```


Dispunerea câmpurilor în structuri

– făcută de compilator în ordinea declarării câmpului
– pot fi însă spaţii goale între câmpuri pentru *alinierea* lor eficientă.

```
struct s {
  char c; // la deplasamentul 0 în structură
  int n;  // poate la deplasament 4 (pe arh. de 32 biţi)
  char a[10];   // poate după el sunt doi octeţi liberi
  double d; // pentru ca d să fie tot la multiplu de 4
} x;
```

Putem afla dispunerea câmpurilor făcând diferenţe intre adresele lor:
`(char *)&x.n - (char *)&x` indică deplasamentul câmpului `n`
(conversie pt. pointeri de acelaşi tip, `char *` pt. dimensiune numerică)

La fel: macro-ul `offsetof`*(tipstruct, numecamp)* (`stddef.h`)
`offsetof(struct s, n)` dă deplasamentul lui `n` în structura `s`

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 !

Compararea zonei de memorie cu `memcmp`: doar dacă structurile sunt iniţializate la fel (spaţiile goale pot avea valori nedeterminate)!

```
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 selecţie:

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

Operatorii `.` şi `->` au precedenţa cea mai ridicată, ca şi `()` şi `[]`
Atenţie 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 şi tablouri

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

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

Ex.: dacă două tablouri au acelaşi domeniu pt. indici şi datele de la acelaşi 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 cu tablou flexibil

Excepţional, ultimul câmp dintr-o structură cu mai multe câmpuri poate fi tablou fără dimensiune specificată.
– dimensiunea structurii e deplasamentul unei structuri identice în care tabloul ar avea elemente (ţine cont de aliniere)
– folosire: prin alocare dinamică, cu lungimea dorită a tabloului:

```
typedef struct s {
  int l;
  char s[];
} string;
string *ps;
char tab[] = "un sir";
int l = strlen(tab);
ps = malloc(sizeof(struct s) + l);
if (ps) {
  ps->l = l;
  memcpy(ps->s, tab, l);
}
```


Structuri de date recursive

Un câmp al unei structuri nu poate fi o structură de acelaşi tip (s-ar obţine o structură de dimensiune infinită/nedefinită!).
Poate fi însă *adresa* unei structuri de acelaşi tip (un pointer)!
⇒ structuri de date recursive, înlănţuite (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 */
```


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 măşti şi utilizarea unor operatori pe biţi
câmp ::= *nume* : *int_const* ; | : *int_const* ;

```
struct packet {
  int : 2;        /* primii doi biţi nu interesează */
  int error: 1;   /* intreg pe un bit: 0 sau 1 */
  int status: 3;  /* intreg pe 3 biţi: 0 .. 7 */
  int : 0;        /* forţează alinierea la octetul următor */
  int seq_no: 4;  /* intreg pe 4 biti: 0 .. 15 */
} 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* conține *toate* câmpurile declarate
– o variabilă *uniune* conține *exact una* din variantele date
(dimensiunea tipului e dată de cel mai mare câmp)

– o variabilă uniune *nu conține* informații 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;
}
```


Definirea tipurilor: observații practice

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

– tablouri: folosiți 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 */ }
```

– concepeți structuri de date ușor de modificat și de extins

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

⇒ definiți pentru acestea cu `typedef` tipuri modificabile ulterior
⇒ folosiți tipurile oferite de limbaj (ex. `size_t`)