Limbaje de programare

# Tipuri definite de utilizator

(structuri, uniuni, enumerări)

8 decembrie 2009

# Tipuri structură

Grupează mai multe elemente de tipuri diferite, legate logic între ele

```
struct lung {   // defineste tipul 'struct lung'
  double val;
  char unit[2];
} v1;           // si declara o variabila de acel tip
struct lung v2 = { 60, "km" }, v3; // decl. cu/fara initializare

struct vect {
  double x, y;
};              // declara tipul struct vect, nu şi variabile
```

Elementele unei structuri se numesc *câmpuri* (engl. fields)
pot fi de orice tip, dar *NU* de *același* tip structură (nu recursiv)

*Folosirea câmpurilor*: cu sintaxa *nume_variabila*.*nume_câmp*
punctul . e *operatorul de selecție* (e un operator postfix)

```
struct vect p1; p1.x = 2; p1.y = 3; printf("%f %f\n", p1.x, p1.y);
```

# Exemplu de structură

```
struct student {     // numele complet al tipului (incl. "struct")
  char nume[32], prenume[32];   // două tablouri de caractere
  char *domiciliu;   // ADRESA! memoria pt. sir se alocă altundeva
  char nr_tel[10];   // max. 9 cifre + terminator \0
  float medie_an[4]; // declaratiile campurilor de structura
  float nota_dipl;   // arata la fel ca declaratiile de variabile
} s;                 // declara variabila s de tip struct student

strcpy(s.nume, "Stefanovici");   // NU! s.nume = ... (e un tablou)
s.domiciliu = "str. Linistei nr. 2";    // sau malloc + strcpy
s.medie_an[2] = 9.35;   // un câmp se foloseşte ca orice variabilă
```

Numele câmpurilor se văd doar în interiorul structurii

⇒ nu putem folosi doar numele cămpului, doar `var.câmp`

⇒ tipuri structuri diferite pot avea câmpuri numite la fel

# Folosirea structurilor (cont.)

Structurile *pot* fi atribuite în totalitatea lor.

```
struct vect v1 = {2, 3}, v2;  v2 = v1;
```

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

```
struct vect add(struct vect v1, struct vect v2)
{
  struct vect v;
  v.x = v1.x + v2.x; v.y = v1.y + v2.y;
  return v;
}
```

Putem scrie *valori compuse* de tip structură, indicând tipul între ( )

```
struct vect v1; v1 = (struct vect){-4, 5};
```

*NU putem compara* structuri cu operatori logici
⇒ trebuie comparate câmp cu câmp: `v1.x==v2.x && v1.y==v2.y`

# Declararea de tipuri

Putem da noi nume la tipuri existente
(mai expresive; sau mai scurte, fără `struct`):

Forma generală: `typedef nume-tip-existent nume-tip-nou;`

Ex. `typedef double real;` `typedef struct vect vect_t;`
(ca declarația de variabile + `typedef` în față ⇒ declară un *tip*)

Numele nou se poate da direct în definirea tipului:

```
typedef struct student { /* ceva campuri */ } student_t;
```

sau separat de definirea tipului structură propriu-zis:

```
typedef struct student { /* ceva campuri */ }; // defineşte tipul
typedef struct student student_t; // declară un nume pentru el
```

# Pointeri la structuri

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

```
struct student *p, s;  p = &s; (*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)++` | (`->` e prioritar) |
| `++p->x` | înseamnă | `++(p->x)` | (`->` e prioritar) |
| `*p->x` | înseamnă | `*(p->x)` | (`->` e prioritar) |
| `*p->s++` | înseamnă | `*((p->s)++)` | (`++` e prioritar lui `*`) |

# Structuri și tablouri

În C, tipurile agregat (compuse) 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.: două tablouri de aceiaș indici, folosite împreună
$\Rightarrow$ înlocuim cu un tablou de element structură:

```
char* nume_luna[12] = { "ianuarie", /* ... , */ "decembrie" };
char zile_luna[12] = { 31, 28, 31, 30, /* ... , */ 30, 31 };
// e preferabilă varianta următoare
struct luna {
  char *nume;
  int zile;
};
struct 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)!
$\Rightarrow$ structuri de date recursive, înlănțuite (liste, arbori, etc.)

```
struct wl {         // struct wl e un tip, incomplet definit
  char *word;       // cuvantul: informaţia propriu-zisă
  struct wl *next; // pointer la structura de acelaşi tip
};                  // acum definiţia tipului e completă
```

Un arbore binar, având în noduri numere întregi:

```
typedef struct t tree; // defineşte tipul incomplet tree = struct t
struct t {
  int val;
  tree *left, *right;  // foloseşte numele din typedef
};           // aici tipul struct t e complet şi echivalent cu tree
```

# Structuri cu câmpuri pe biţi

Vrem să reprezentăm mai multe informaţii cât mai compact pe biţi.
Ex.: o dată ca întreg pe 32 de biţi: sec, min (0-59): 6 biţi, ora (0-23), ziua (1-31): 5 biţi, luna (1-12): 4 biţi), anul (1970 + 0-63): 6 biţi.
Construim : `int data = 39 << 26 | 5 << 22 | 19 << 17 | 17 << 12;`
(19.5.2009, 17h). Extragem ora: `int ora = data >> 12 & 0x1F;`

Sau: acces direct la câmpuri pe biţi, fără măști și operatori pe biţi.

```
struct date_t {    // alternativa: structură cu câmpuri pe biţi
  unsigned sec, min : 6;  // indică numărul de biţi
  unsigned hour, day: 5;  // se permit tipuri întregi
  unsigned month: 4;
  unsigned year: 6;
} data = {0, 0, 17, 19, 5, 39 };    // 17:00:00, 19.05.(1970+39)
```

Putem scrie direct: `printf("%u.%u\n", data.day, data.month);`

Putem avea câmpuri fără nume: `int:  2; // pe 2 biţi`
sau forţa trecerea la memorarea în octetul următor `int:  0;`

# Tipul enumerare

*Tipul enumerare*: dă nume unui șir de valori numerice.
⇒ folosit când e mai sugestiv de scris un nume decât un număr

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

definește un tip cu numele `enum luni_curs` (și `enum` *e parte din nume*!)
Implicit, șirul valorilor e crescător începând cu 0
dar putem specifica și explicit valori (și o valoare se poate repeta)
Un tip enumerare e un tip întreg
⇒ variabilele/valorile enumerare se folosesc ca și întregi

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

Un nume de constantă nu poate fi folosit în mai multe enumerări

# Uniuni

Folosite pentru a reține valori care pot avea tipuri *diferite*.
Sintaxa: ca la structuri, dar cu cuvântul cheie `union`
Lista de câmpuri reprezintă o listă de variante, pentru fiecare tip:
– 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)

```
union {          // tip uniune, fara nume
  int i;
  double r;
  char *s;
} val;           // trei variante pentru fiecare tip de valoare
enum { INT, REAL, SIR } tip;   // tine minte varianta memorata
char s[32]; if (scanf("%31s", s) == 1) {
  if (isdigit(*s))  // incepe cu cifra ? daca da, contine punct ?
    if (strchr(s, '.')) { sscanf(s, "%lf", &val.r); tip = REAL; }
    else { sscanf(s, "%d", &val.i); tip = INT; }
  else { val.s = strdup(s); tip = SIR; }
}
```