

# Tipuri definite de utilizator

4 decembrie 2003

## Generalități

---

Un tip definește o *mulțime de valori* și *operațiile* posibile cu acestea. Adeseori e nevoie de alte tipuri (mai complexe) decât cele de bază. În C se pot defini tipuri enumerare, structură și uniune cu sintaxa:

*cuvânt\_cheie* *opt\_nume\_tip* *specificatie\_tip* *opt\_lista\_declaratori* ;

unde *cuvânt\_cheie* ::= `enum` | `struct` | `union`

*opt\_nume\_tip*: pt. referire ulterioară (prefixat cu `enum`, `struct`, `union`)

*opt\_lista\_declaratori*: pot fi declarate obiecte de tipul respectiv

– *opt\_nume\_tip* e într-un spațiu de nume diferit de cel comun pentru variabile, tipuri și funcții. E mai clar să folosim totuși nume diferite.

## Tipuri enumerare

---

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

Sintaxa: `enum opt_nume_tip { lista_constante } opt_lista_declaratori ;`

– 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

– același nume de constantă nu poate fi folosit în două enumerări diferite

– tipurile enumerare sunt tipuri întregi

⇒ variabilele enumerare se pot folosi la fel cu variabilele întregi

– cod mai lizibil decât prin declararea separată de constante

```
int ore_lucru[7];
```

```
enum zile_sapt {D, L, Ma, Mc, J, V, S} zi;
```

```
for (zi = L; zi <= V; zi++) ore_lucru[zi] = 8;
```

## Structuri

---

Folosite pentru gruparea mai multor elemente de tipuri de date diferite  
– exemplu clasic: înregistrare din bază de date cu 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 opt_nume_tip { lista_câmpuri } opt_lista_declaratori ;
```

- oricare poate lipsi, dar nu amândouă (tipul n-ar putea fi folosit)
- elementele unei structuri se numesca *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)

## Definirea tipurilor: observații practice

---

- definirea unor nume de tip (`typedef`) facilitează înțelegerea codului
- tablouri: constante simbolice (nu direct numere) pentru dimensiune (modificările ulterioare sunt necesare într-un singur punct în program)
- 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.

## Utilizarea structurilor

---

Accesul la câmpuri: se face cu sintaxa *nume\_variabila.nume\_câmp* (operația de *selectie*)

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

Inițializarea structurilor: câmp cu câmp, între acolade, ca și pentru alte agregate: `struct coord { float x, y; } punct1 = { 2.5, 1.5 };`

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

NU se compară structuri cu operatori logici (doar pt. câmpuri), sau:

```
int memcmp(void *p1, void *p2, size_t n); /* in stdlib.h */  
struct { ... } x, y;    if (memcmp(&x, &y, sizeof(x))) ...
```

## Pointeri la structuri

---

Accesul la câmpuri se face frecvent printr-un pointer la structură:

```
struct student s, *p; 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)++`

`++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 {          /* o listă de cuvinte */
    char *word;      /* informația propriu-zisă */
    struct wl *next; /* pointer la același tip de structură */
};
```

Un arbore binar, având în noduri numere întregi:

```
typedef struct t tree; /* declarație incompletă */
struct t {
    int val;
    tree *left, *right; /* folosește numele din typedef */
};
```

## Câmpuri pe biți

---

Se pot declara câmpuri întregi cu un număr specificat de biți

⇒ Testarea/setarea unor biți se face folosind direct numele câmpului fără a fi nevoie de definirea de măști și utilizarea unor operatori pe biți

*câmp ::= nume : int\_const ; | : int\_const ;*

```
struct packet {
    int : 2;          /* primii doi biți nu interesează */
    int error: 1;    /* un bit, semnalizează eroare */
    int status: 3;   /* un câmp pe 3 biți */
    int : 0;         /* forțează alinierea la octetul următor */
    int seq_no: 4;   /* număr de secvență pe 4 biți */
} 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)

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;    /* șir 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;
}
```