Programarea calculatoarelor

Tipuri definite de utilizator

19 mai 2009

Generalități. Tipul enumerare

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

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

Tipuri structură

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

```
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
} stud1, stud2;      // două variabile declarate direct aici
struct student s3;   // declaraţie separată de variabilă
```

Elementele unei structuri se numesc *câmpuri* (engl. fields)

pot fi de orice tip, dar *NU* de *același* tip structură (nu recursiv)

Numele câmpurilor se văd doar în interiorul structurii

$\Rightarrow$ tipuri structuri diferite pot avea câmpuri numite la fel

Folosirea structurilor. Operatori

1) toată structura (atribuite, date ca parametru, returnate ca rezultat)
2) accesul la câmpuri: se face cu sintaxa *nume_variabila*.*nume_câmp*
Punctul . e *operatorul de selecţie* (e un operator postfix)

```
struct student s; // var. s; numele complet al tipului: 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
```

Iniţializarea structurilor: câmp cu câmp, cu acolade, ca la tablouri

```
struct point { float x, y; } pct1 = { 2.5, 1.5 };
```

Putem scrie direct valori de tip structură (*literale compuse*)

(*nume-tip*) { *iniţializatori* } // tip indicat explicit în conversie

```
void draw(struct point p);      // declaraţie de funcţie
struct point p2;                // declară o variabilă structură
p2 = (struct point) { -1, 2 };  // indică ce tip are valoarea dată
draw((struct point) { 1.5, 2.5 }); // struct. dată ca parametru
```

Folosirea 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

Declararea de tipuri: `typedef nume-tip-existent nume-tip-nou;`

ex. `typedef double real;`

⇒ putem alege un nume mai scurt, fără `struct`.

– 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; /* 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
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 măşti şi 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;`

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; }
}
```

Programe compuse din mai multe fişiere

Programele se scriu modular, într-un fişier `.c` se pun funcţii înrudite (exemplu: un fişier *bibliotecă* de funcţii pentru lucrul cu un anumit tip)

Declaraţiile de tipuri, funcţii şi variabile care trebuie folosite şi în alte fişiere se pun într-un fişier antet `.h`
Acesta e inclus de fiecare fişier `.c` care îl necesită.

Fişierul `biblioteca.c` poate fi *compilat separat* (`gcc -c`) într-un *fişier obiect* `.o` – acesta conţine cod maşină, dar cu informaţii despre numele de funcţii/variabile, pentru a putea fi folosite

Programatorul scrie un program `main.c` care include `biblioteca.h` şi e compilat împreună cu codul bibliotecii (e suficient fişierul obiect).

Tipuri de date abstracte

TDA = un model matematic cu un set de operaţii asupra lui
⇒ o structură de date + funcţii care operează pe ea
⇒ noţiunea de *clasă* din programarea orientată pe obiecte

Pentru implementarea TDA în C:
– în fişierul `.h` se declară minimul necesar pentru a putea compila programul (pentru structuri, adesea doar un `typedef` pt. pointer la tip)
– şi declaraţii de funcţii care manipulează tipul respectiv
– structura tipului şi definiţiile funcţiilor: ascunse în implementare (`.c`)

```
typedef struct node *list_t; /* în fişierul .h */
typedef struct node {        /* în fişierul .c cu implementarea */
  int info;                  /* sau/şi alte câmpuri */
  struct node *nxt;
} node_t;                    /* tip vizibil doar în fişierul .c */
```

Utilizatorul, care include doar fişierul `.h` nu are acces la structura internă a tipului (`node_t`); accesul e permis doar prin funcţii.
⇒ schimbări în implementarea bibliotecii nu afectează programul.