Alocare dinamică. Pointeri la funcţii

16 mai 2006

Pointeri: recapitulare

– o variabilă pointer contine o *adresă*

– în program, dorim să prelucrăm *obiecte* (numere, şiruri, etc.)

– declarând un pointer *tip* `*p` avem loc pentru memorarea *adresei* unui obiect de tipul *tip*, *NU* şi a obiectului propriu-zis !

– putem indica prin pointeri doar obiecte deja declarate în program.

```
int x; int *p; p = &x; // p ia adresa unei variabile existente
char a[20]; char *s; s = a+5; // sau s = &a[5];
```

⇒ până acum am folosit doar variabile alocate *static* (la compilare)

⇒ dimensiunea acestora e cunoscută la compilare
(şi se poate afla în program cu operatorul `sizeof`)

– problema: nu putem declara variabile cu dimensiuni cunoscute doar la rulare (de ex. un tablou cu lungime dată la intrare de utilizator)

Alocarea dinamică (funcţii din `stdlib.h`)

pt. gestionarea memoriei după nevoile ce apar la *rularea* programului

`void *calloc(size_t num, size_t size);`
alocă `num * size` octeţi iniţializaţi cu 0

`void *malloc(size_t size);` alocă `size` octeţi, *neiniţializaţi*!

`void *realloc(void *ptr, size_t size);` realocă la dimens. `size`
creşte/scade blocul de la adresa `ptr`, alocat anterior *tot dinamic*
poate muta blocul; păstrează conţinutul pe $\min(\text{dim}_{\text{veche}}, \text{dim}_{\text{noua}})$

Toate 3 returnează adresa alocată sau `NULL` (eroare, mem. insuficientă)
$\Rightarrow$ e obligatorie testarea valorii returnate !

`void free(void *ptr);` eliberează memoria alocată cu c/m/realloc

```
int i, n, *t; // citire tablou cu număr indicat de elemente
printf("Nr. de elemente ?"); scanf("%d", &n);
if ((t = malloc(n * sizeof(int)) != NULL)
  for (i = 0; i < n; i ++) scanf("%d", &t[i]);
```

Când și cum folosim alocarea dinamică ?

NU e necesară când știm de la început de câtă memorie e nevoie; NU:

```
int *px; px = malloc(sizeof(int)); scanf("%d", px); printf("%d", *px); free(px);
```

```
char *s = malloc(80); scanf("%79s", s); puts(s); free(s);
```

ci simplu: `int x; scanf("%d", &x); char s[80]; scanf("%79s", s);`

DA, când nu stim de la compilare câtă memorie e necesară
(tablouri cu dimensiune aflată la rulare, liste, arbori, etc.)
DA, când trebuie să returnăm memorie nou creată dintr-o funcţie
(NU putem returna adresa unei structuri locale: dispare după apel !!)

```
char *strdup(const char *s) { // crează copie a lui s)
  char *d = malloc(strlen(s) + 1); //nu: char d[strlen(s)+1];
  return d ? strcpy(d, s) : NULL;
}
```

Folosim `malloc/calloc` când putem calcula direct necesarul de memorie
NU folosim inutil `realloc` în mod repetat când putem calcula totalul

Exemplu: citirea unei linii de dimensiune nelimitată

```
#include <stdio.h>
#include <stdlib.h>
const int ADD = 16;
char *getline(void) {
  char *p, *s = NULL; // realloc(NULL, sz) e ca si malloc(sz)
  int c, lim = -1, size = 0;  // limita si dimensiune curenta
  while ((c = getchar()) != EOF) {
    if (size >= lim) // (re)alocă memorie, testează de eroare
      if (!(p = realloc(s, (lim+=ADD)+1))) { // creste limita
        ungetc(c, stdin); break;  // nu mai avem loc
      } else s = p; // succes -> foloseste noul pointer
    if ((s[size++] = c) == '\n') break; // linie noua -> gata
  }  // trunchiaza apoi linia la dimensiunea necesara
  if (s) { s[size++] = '\0'; realloc(s, size); }
  return s;
}
```

Pointeri la funcţii

Adresa unei funcţii se poate obţine, memora, şi utiliza pentru a o apela.
pentru o funcţie *tip_rez* `fct (`*tip1*`, ..., `*tipn*`);`
adresa are tipul *tip_rez* `(*pfct) (`*tip1*`, ..., `*tipn*`);`
se poate atribui `pfct = fct;` (numele funcţiei reprezintă adresa ei)

Tipul unui pointer la funcţie e dat de tipul rezultatului şi al parametrilor

Atenţie! `int *fct(void);` funcţie ce returnează pointer la întreg
`int (*fct)(void);` pointer la o funcţie ce returnează întreg

Exemplu: selecţia unor prelucrări în funcţie de tasta apăsată

```
void help(void) {/*...*/} void menu(void) {/*...*/}
/* alte functii de acelasi tip */ void quit(void) {/*...*/}
void (*ftab)[10](void) = { help, menu,/*etc,*/quit }; //tab.de pointeri
void do_cmd(void) {  // citeste o tasta si executa o functie
  int k = getchar() - '0'; // valoarea, daca e cifra
  if (k >= 0 && k <= 9) ftab[k](); // apeleaza fct k din tab.
} // mai concis decât if (k==0) help(); else if (k==1) menu(); else ...
```

Pointeri la funcţii: prelucrare parametrizată

```
int add1(int x) { return x+1; } int mul2(int x) { return 2*x; }
void map(int tab[], int len, int (*pf)(int)) {
  for (int i = 0; i < len; ++i) // parcurge tabloul
    tab[i] = pf(tab[i]); // si aplica functia data parametru
}
int main(void) {
  int t[] = {  4, 3, 5, 2 };
  map(t, 4, mul2); // parcurge si aplica mul2: { 8, 6, 10, 4 }
  map(t, 4, add1); // parcurge si aplica add1: { 9, 7, 11, 5 }
}
```

Algoritmul *quicksort*, declarat (în `stdio.h`) ca funcţie cu parametrii:

```
void qsort(void *tab, size_t num, size_t sz, int (*cmp)(const void *, const void *));
```

– adresa tabloului de sortat, numărul şi dimensiunea elementelor

– adresa funcţiei care compară 2 elemente (returnează $<$, $=$ sau >0)

efectuarea comparării depinde de tip: întreg, şir, definit de utilizator

– foloseşte argumente `void *` fiind compatibile cu pointeri la orice tip

– în corpul funcţiei de comparare, forţăm pointerii la tipul din problema cu operatorul de conversie de tip: (*tip*)*expresie*

Exemplu: alocare dinamică + sortare

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 100 // alocăm pt. 100 de numere odată
int cmp(const void *p, const void *q) {
  return *(int *)p - *(int *)q; } // fortam p si q la tipul int *
}  // si facem diferenta intre valorile intregi de la adresele p si q
int main(void) { // sorteaza intregii cititi pana introducem zero
  int i = 0, n = 0, *t = NULL, *t1; // contor, total, tablou, temp
  do { // alocă câte INCR întregi, iniţial şi când e nevoie
    if (i == n) { // initial, realloc(NULL,sz) e ca malloc(sz)
      n += INCR;
      if (t1 = realloc(t, n*(sizeof(int)))) t = t1; // succes
      else { printf("nu mai avem loc!\n"); break; }
    }
    if (scanf("%d", &t[i]) != 1) return -1; // iese la eroare
  } while (t[i++]); // conventie: citim până introducem zero
  qsort(t, i, sizeof(int), cmp);
  for (n = 0; n < i; n++) printf("%d ", t[n]);
  free(t); // nu uitam sa eliberam memoria !
  return 0;
}
```

Instrucţiuni etichetate

identificator `:` *instrucţiune*
`case` *expresie-constantă-int* `:` *instrucţiune*
`default :` *instrucţiune*

– permit referirea la instrucţiune pentru un salt (explicit sau implicit).
Etichetele au *spaţiu de nume* separat de cel al identificatorilor obișnuiți (putem avea variabile/funcţii/etc. și etichete cu același nume)

Domeniul de vizibilitate al etichetei e corpul funcţiei în care se află (numele de etichete trebuie să fie unice în cadrul unei funcţii)

Etichetele `case` și `default` pot apare doar în instrucţiunea `switch`.
Într-o instrucţiune `switch` poate exista cel mult o etichetă `default` iar constantele întregi din etichetele `case` vor fi distincte.

Instrucţiunea `switch`

`switch` (*expresie-intreagă*) *instrucţiune*

– se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)

– dacă în corpul *instrucţiune* (compusă) există o etichetă **`case`** cu valoarea întreagă obţinută, se sare la instrucţiunea respectivă

– dacă nu, şi există o etichetă **`default`**, se sare la acea instrucţiune

– altfel nu se execută nimic (se trece la instrucţiunea următoare)

– pt. acelaşi cod la mai multe etichete: **`case`** *val1* **`:`** **`case`** *val2* **`:`** *şir-instr*

Obs: Execuţia nu se opreşte la următorul **`case`** (e doar o etichetă); ieşirea din **`switch`**: doar cu instruct. **`break`** sau la sfârşitul corpului!

$\Rightarrow$ permite utilizarea de cod comun pe ramuri, dar cu mare atenţie!

Instrucţiunea `switch`: exemplu

```
char c; int a, b, r;
printf("Scrieti o operatie intre doi intregi: ");
if (scanf("%d %c %d", &a, &c, &b) == 3) { /* toate 3 corect */
  switch (c) {
    case '+': r = a + b; break;  // iese din corpul switch
    case '-': r = a - b; break;  // idem
    default: c = '\0'; break;    // fanion pt. caracter eronat
    case 'x': c = '*';  // considera 'x' tot înmulţire, continuă
    case '*': r = a * b; break;  // executat pt. x si *, apoi iese
    case '/': r = a / b;    // la sfârşit nu e necesar break
  }
  if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
  else printf("Operaţie necunoscută\n");
} else printf("Format eronat\n");
```