

## Alocare dinamică. Pointeri la funcții

16 mai 2006

Programarea calculatoarelor. Curs 11

Marius Minea

Alocare dinamică. Pointeri la funcții

2

### Pointeri: recapitulare

– o variabilă pointer conține o *adresă*  
– în program, dorim să prelucrăm *obiecte* (numere, șiruri, etc.)  
– declarând un pointer *tip* \*p avem loc pentru memorarea *adresei* unui obiect de tipul *tip*, *NU* și 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  
(și 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)

Programarea calculatoarelor. Curs 11

Marius Minea

Alocare dinamică. Pointeri la funcții

3

### Alocarea dinamică (funcții din `stdlib.h`)

pt. gestionarea memoriei după nevoile ce apar la *rularea* programului  
`void *calloc(size_t num, size_t size);`  
alocă `num * size` octeți inițializați cu 0  
`void *malloc(size_t size);` alocă `size` octeți, *neinițializați!*  
`void *realloc(void *ptr, size_t size);` realocă la dimensiune `size`  
crește/scade blocul de la adresa `ptr`, alocat anterior *tot dinamic*  
poate muta blocul; păstrează conținutul pe `min(dim_veche, dim_noua)`  
Toate 3 returnează adresa alocată sau `NULL` (eroare, mem. insuficientă)  
⇒ 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]);`

Programarea calculatoarelor. Curs 11

Marius Minea

Alocare dinamică. Pointeri la funcții

4

### 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 știm 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 funcție (NU putem returna adresa unei structuri locale: dispăre 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 `realloc` în mod repetat când putem calcula totalul

Programarea calculatoarelor. Curs 11

Marius Minea

Alocare dinamică. Pointeri la funcții

5

### 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))) { // crește limita
                ungetc(c, stdin); break; // nu mai avem loc
            } else s = p; // succes -> folosește noul pointer
        if ((s[size++] = c) == '\n') break; // linie nouă -> gata
    } // trunchiază apoi linia la dimensiunea necesară
    if (s) { s[size++] = '\0'; realloc(s, size); }
    return s;
}
```

Programarea calculatoarelor. Curs 11

Marius Minea

Alocare dinamică. Pointeri la funcții

6

### Pointeri la funcții

Adresa unei funcții se poate obține, memora, și utiliza pentru a o apela.  
pentru o funcție `tip_rez fct (tip1, ..., tipn);`  
adresa are tipul `tip_rez (*pfct) (tip1, ..., tipn);`  
se poate atribui `pfct = fct;` (numele funcției reprezintă adresa ei)

Tipul unui pointer la funcție e dat de tipul rezultatului și al parametrilor  
Atenție! `int *fct(void);` funcție ce returnează pointer la întreg  
`int (*fct)(void);` pointer la o funcție ce returnează întreg

Exemplu: selecția unor prelucrări în funcție 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) { // citește o tasta si execută o funcție`  
`int k = getchar() - '0'; // valoarea, dacă e cifră`  
`if (k >= 0 && k <= 9) ftab[k](); // apelează fct k din tab.`  
`} // mai concis decât if (k==0) help(); else if (k==1) menu(); else ...`

Programarea calculatoarelor. Curs 11

Marius Minea

## Pointeri la funcții: 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 funcție cu parametrii:

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

- adresa tabloului de sortat, numărul și dimensiunea elementelor
- adresa funcției care compară 2 elemente (returnează `<`, `=` sau `>` 0) efectuarea comparării depinde de tip: întreg, șir, definit de utilizator
- folosește argumente `void *` fiind compatibile cu pointeri la orice tip
- în corpul funcției de comparare, forțăm 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, inițial și 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;
}
```

## Instrucțiuni etichetate

```
identificator : instrucțiune
case expresie-constantă-int : instrucțiune
default : instrucțiune
```

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

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

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

## Instrucțiunea switch

```
switch ( expresie-intreagă ) instrucțiune
```

- se evaluează expresia (de tip întreg, posibil limitată la 1023 valori)
  - dacă în corpul *instrucțiune* (compusă) există o etichetă `case` cu valoarea întreagă obținută, se sare la instrucțiunea respectivă
  - dacă nu, și există o etichetă `default`, se sare la acea instrucțiune
  - altfel nu se execută nimic (se trece la instrucțiunea următoare)
  - pt. același cod la mai multe etichete: `case val1:` `case val2:` *șir-instr*
- Obs: Execuția nu se oprește la următorul `case` (e doar o etichetă); ieșirea din `switch`: doar cu `instruct. break` sau la sfârșitul corpului!  
⇒ permite utilizarea de cod comun pe ramuri, dar cu mare atenție!

## Instrucțiunea 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 înmulțire, continuă
        case '*': r = a * b; break; // executat pt. x si *, apoi iese
        case '/': r = a / b; // la sfârșit nu e necesar break
    }
    if (c) printf("Rezultatul: %d %c %d = %d\n", a, c, b, r);
    else printf("Operație necunoscută\n");
} else printf("Format eronat\n");
```