

Alocarea dinamică

Folosim *adrese* pentru a lucra de fapt cu *obiectele* indicate prin adresă **ATENȚIE!** declarând un pointer *tip* *p avem loc doar pentru o *adresă*, NU și pentru un *obiect* (variabilă) de *tip*.

Declararea lui `char *s`; NU înseamnă și loc pentru a citi/memora un șir!

Până acum am indicat prin pointeri doar variabile deja declarate:

```
int x; int *p; p = &x; char a[20]; char *s; s = a+5; // s = &a[5];
```

Am declarat *static* doar tablouri de dimensiuni cunoscute și fixe

(în C99 se permit dimensiuni variabile, evaluate la rulare)

Nu putem *crea și returna* dintr-o funcție un tablou: el trebuie declarat în afara funcției, și adresa transmisă la funcție care îl completează (ex. `scanf`, `strcpy`, funcțiile scrise pentru lucrul cu vectori/matrici)

Funcțiile de *alocare dinamică* (`stdlib.h`) permit să creem variabile noi de dimensiuni necesare apărute la *rularea* programului

Programarea calculatoarelor

Alocare dinamică. Pointeri la funcții

Marius Minea

6 mai 2008

Funcții de alocare dinamică (`stdlib.h`)

```
void *malloc(size_t size);          alocă size octeți
void *calloc(size_t num, size_t size); num*size octeți init. Cu 0
- returnează adresa de început unde a fost alocat nr. dat de octeți
sau NULL la eroare (ex. mem. insuficientă) => trebuie testat rezultatul!
modificarea dimensiunii unei zone alocate cu c/malloc:
void *realloc(void *ptr, size_t size); modifică mărimea la size
- poate returna alta adresa decât ptr, atunci mută conținutul existent
=> Ex. if (p1 = realloc(p, size)) { p = p1; /* apoi folosim p */ }
Memoria alocată dinamic trebuie eliberată când nu mai e necesară
void free(void *ptr); eliberează memoria alocată cu c/malloc
int i, n, *t;
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 dinainte de câtă memorie e nevoie

```
NU: int *px; px = malloc(sizeof(int)); scanf("%d", &px);
```

Mai simplu: `int x; scanf("%d", &x);`

DA, când nu știm de la compilare câtă memorie e necesară (tablouri cu dimensiuni aflate la rulare, liste, arbori, etc.)

DA, când trebuie să returnăm un obiect nou creat dintr-o funcție

(NU putem returna adresă de var. locală, memoria dispare la revenire!)

```
char *strdup(const char *s) { // creeaza copie a lui s
    char *d = malloc(strlen(s) + 1); // loc pentru sir si '\0'
    return d ? strcpy(d, s) : NULL; // fa copia, returneaza d
}
```

DA, când trebuie păstrat un obiect citit într-un loc temporar

```
char *tab[10], buf[81];
```

```
while (i < 10 && fgets(buf, 81, stdin))
```

```
    tab[i++] = strdup(buf); // salveaza adresa copiei
```

Pointeri la funcții

Parametrii și **variabilele** ne permit mai mult decât calcule cu valori fixe => uneori dorim să variem *funcția* apelată într-un punct de program

Exemplu: parcurgerea unui tablou pentru diverse prelucrări

```
for (int i = 0; i < len; ++i) f(tab[i]); // (pt. diverse funcții f)
```

=> se poate, folosind variabile *pointeri la funcții*

Numele unei funcții reprezintă chiar *adresa* funcției.

Declaratii: de *funcție*: `tip_rez fct (tip1, ..., tipn);`

de *pointer la funcție* (de același tip): `tip_rez (*pfct) (tip1, ..., tipn);`

se poate atribui `pfct = fct;` (numele funcției reprezintă adresa ei)

Exemplu: `int fct(void);` declară o *funcție* ce returnează un întreg

`int (*fct)(void);` declară un *pointer la o funcție* ce returnează întreg

ATENȚIE! `int *fct(void);` e o funcție ce returnează *pointer la întreg*

Sintaxa pointerilor de funcții e complicată => e util să declarăm un tip:

```
typedef void (*funptr)(void); // tip pointer la funcție void
funptr funtab[10]; // tablou de pointeri de funcție void
```

Utilizarea pointerilor la funcții

```
void mul3(int *p) { *p *= 3; }
void tip(int *p) { printf("%d ", *p); }
void prel(int tab[], int len, void (*fp)(int *p)) {
    for (int i = 0; i < len; ++i) fp(&tab[i]);
} // apoi in main putem scrie:
```

```
int t[LEN] = { 2, 3, 5, 7, 11 }; // tabloul de prelucrat
```

```
prel(t, LEN, mul3); /*inmulteste*/ prel(t, LEN, tip); //afiseaza
```

Exemplu: funcția standard de sortare **qsort** (`stdlib.h`)

```
void qsort(void *base, size_t num, size_t size, int (*comp)(void *, void *));
```

- adresa tabloului de sortat, numărul și dimensiunea elementelor

- adresa funcției care compară 2 elemente (returnează <, = sau > 0)

=> folosește argumente `void *` fiind compatibile cu pointeri la orice tip

```
typedef int (*comp_t)(const void *, const void *); //tip ptr.fct.comp
```

```
int intcmp(int *p1, int *p2) { return *p1 - *p2; } //fct.comp.intregi
```

```
int tab[5] = { -6, 3, 2, -4, 0 }; // tabloul de sortat
```

```
qsort(tab, 5, sizeof(int), (comp_t)intcmp); // sorteaza crescator
```