Pointeri

2 noiembrie 2005

Variabile şi adrese

În limbajul C, orice variabilă are o *adresă*: o valoare numerică; indică locul din memorie unde e memorată valoarea variabilei

Operatorul prefix `&` dă adresa operandului: `&x` e adresa variabilei `x`
Operandul: un *lvalue* = orice poate apare la stânga unei atribuiri (variabilă, element de tablou, funcţie; NU pentru expresii oarecare)

Poate fi tipărită (în hexazecimal) cu formatul `%p` în `printf`
Adresele sunt întregi, dar posibil ≠ `sizeof(int)` (atenţie la conversie!)

```
#include <stdio.h>
double d; int a[10]; // variabile globale, in zona de date
int main(void)
{
  int k;             // variabilă locală, pe stivă (altă zonă)
  printf("Adresa lui d: %p\n", &d); /* de ex. 0x80496c0 */
  printf("Adresa lui a[0]: %p\n", &a[0]);  /* 0x80496e0 */
  printf("Adresa lui a[5]: %p\n", &a[5]);  /* 0x80496f4 */
  printf("Adresa lui k: %p\n", &k);        /* 0xbffff8e4 */
} // &a[5] - &a[0] == 5 * sizeof(int)  (poziţii consecutive)
```


Tipuri pointer. Declarare. Indirectare

Orice expresie în C are un tip ⇒ la fel şi expresiile adresă.
OBS: Dacă variabila `x` are tipul `tip`, `&x` are tipul `tip *`
`int x;` ⇒ `&x` are tipul `int *`, adică pointer la `int` (adresă de `int`)
`char c;` ⇒ `&c` are tipul `char *`, (pointer la `char`, adresă de `char`)
⇒ există tipuri de adresă diferite pentru fiecare tip de date
⇒ putem *declara* variabile de aceste tipuri (pointeri):

tip `*` *nume_var*; *nume_var* e pointer la (adresă pt.) o valoare de *tip*
pointer = o variabilă care conţine *adresa* altei variabile

Operatorul prefix `*` dă obiectul `*p` de la adresa dată de operandul `p`
Operand: pointer. Rezultat: *referinţă* la obiectul indicat de pointer
⇒ operator de *indirectare* (dereferenţiere, referire indirectă prin adresă)
OBS: Dacă pointerul `p` are tipul `tip *`, `*p` are tipul `tip`

Sintaxa declaraţiei (aceeaşi dar citită în două feluri) sugerează folosirea:
`char*  p;` `p` e o variabilă de tipul `char *` (adresă de `char`)
`char  *p;` `*p` (obiectul de la adresa `p`) are tipul `char`

Operatorii de adresă şi dereferenţiere

Pointerii au adrese, ca orice variabile:
pt. `int *p;` adresa `&p` are tipul `int **`
adică adresa unei adrese de `int`
Putem citi `int* *pp` sau `int **pp`, deci:
`*pp` are tipul `int *` (val. `p`, adresă de `int`)
`**pp` are tipul `int` (val. `x`, de la adr. `*pp`)

	Variabilă	Valoare	Adresă
`int x=5;`		5	0x408
		...	
`int *p=&x;`		0x408	0x51C
		...	
`int **pp=&p;`		0x51C	0x9D0

Înainte de folosire, un pointer trebuie *iniţializat*, de ex. cu adresa unei variabile de tipul potrivit: `int x, *p, **pp; p = &x; pp = &p;`

O *referinţă* `*p` este un obiect care se poate folosi si la stanga unei atribuiri (*lvalue*), ca o variabilă (evident si la dreapta, ca orice expresie).
In cazul de mai sus, `*p` se foloseşte absolut la fel (sinonim) cu `x`:
`int x, y, z, *p; p = &x; z = *p; /* z = x */ *p = y; /* x = y */`

OBS: Operatorii adresă `&` şi de indirectare `*` sunt *unul inversul celuilalt*:
`*&x` este chiar `x`, pentru orice obiect (variabilă) `x`
`&*p` are valoarea `p`, pentru orice variabilă pointer `p`
(dar `p` e o *variabilă* şi poate fi atribuită; `&*p` e o *expresie* şi nu poate)

Tipuri şi valori pentru pointeri

Doi pointeri spre tipuri diferite au tipuri diferite. `char * ≠ int *`
(Nu se pot atribui între ei, direct sau transmişi ca argumente la funcţii)

Toţi pointerii au valori adrese ⇒ `sizeof(int *) == sizeof(float *)` etc.
⇒ putem face *conversie explicită* `( )` între două tipuri pointer
(dăm o altă interpretare octeţilor de memorie la acea adresă)

```
int x; char *pc = (char *)&x;      *pc conţine primii 8 biţi (inferiori) din x
char s[20]; int n = *(int *)s;
   n conţine un întreg cu biţii din primii sizeof(int) caractere din s
float f; long n; n = *(long *)&f;
   n are aceiaşi biţi ca si f, dacă sizeof(float) == sizeof(long)
```

Tipul `void *` e folosit ca tip de adresă generică (nu indică nimic)
– poate fi atribuit în ambele sensuri la orice alt pointer, fără `( )`
– nu poate fi dereferenţiat fără conversie (nu ştim ce indică)

Trebuie indicat când se cere un pointer dar nu putem da o adresă validă
⇒ `NULL` definit în `stddef.h` ca `(void *)0` == adresă distinctivă invalidă
(zona de memorie de la adresa 0 nu a accesibilă programului)

Erori în lucrul cu pointeri

Utilizarea *oricărei* variabile neiniţializate e o eroare logică în program !
`{ int x; printf("%d", x); } // cât e x ?? valoare la întâmplare!`

Pointerii trebuie iniţializaţi înainte de folosire, ca orice variabile!
– cu adresa unei variabile (sau cu alt pointer iniţializat deja)
– cu o adresă de memorie alocată dinamic (vom discuta ulterior)

EROARE: *tip* `*p;` `*p` = *valoare*; `p` este *neiniţializat*!! (eventual nul)
⇒ valoarea va fi scrisă la o adresă de memorie necunoscută (evtl. nulă)
⇒ coruperea memoriei, rezultare eronate sau imprevizibile, terminarea forţată a programului (sub sisteme de operare cu memorie protejată)

La fel: *orice parametru* de funcţie trebuie să aibă *o valoare definită*.
Greşit: `char *p; scanf("%s", p);` Corect: `char p[10]; scanf("%9s", p);`
⇒ valoare definită pentru `p` + citire limitată la memoria alocată.
conţinutul de la adresa `p` poate fi neiniţializat, dacă funcţia *scrie* acolo şi nu citeşte; dar *valoarea* adresei trebuie să indice memorie existentă

Verificaţi că expresiile au tipuri compatibile (ex. la atribuire)
⇒ valabil şi pentru pointeri (nu confundaţi `p` cu `*p`, etc.)

Pointeri ca argumente/rezultate de funcţii

Permit modificarea valorii unei variabile prin transmiterea adresei ei
– o variabilă se poate modifica prin indirectarea unui pointer către ea
– nu se modifică *adresa* (transmisă tot prin valoare) ci *conţinutul* ei

```
void swap (int *pa, int *pb) /* schimbă val. de la adr. pa şi pb */
{
  int tmp; /* variabilă auxiliară necesară pentru interschimbare */
  tmp = *pa; *pa = *pb; *pb = tmp;  /* trei atribuiri de întregi */
} /* în funcţie s-a lucrat cu conţinutul de la adresele pa şi pb */
```

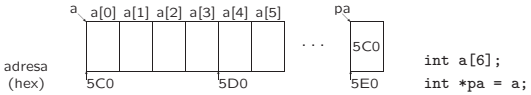
Ex.: `int x = 3, y = 5; swap(&x, &y); /* acum x = 5 şi y = 3 */`
OBS: Nu se poate obţine efectul cu `void swap(int m, int n);`
(ar schimba valorile transmise în corpul funcţiei, fără efect în afară)

Folosire: când limbajul nu permite transmiterea prin valoare (*tablouri*) sau ar fi ineficientă (structuri mari) ⇒ transmitem *adresa* variabilei

Tablouri şi pointeri

În limbajul C noţiunile de *pointer* şi *nume de tablou* sunt asemănătoare.
Declararea unui tablou alocă un bloc de memorie pt. elementele sale
Oriunde (excepţie: în `sizeof`), *numele tabloului e adresa* acestui bloc
⇒ pentru tabloul `tip a[LEN];` numele `a` e o *constantă* de tipul `tip *`
`&a[0]` e echivalent cu `a` (adresa tabloului e adresa primului element)
`a[0]` e echivalent cu `*a` (obiectul de la adresa `a` e primul element)
În general: `&a[i]` e definit ca `a+i` şi `a[i]` definit ca `*(a+i)`

Dacă declarăm `tip *pa;` putem atribui `pa = a;`
Diferenţa: adresa `a` e o *constantă* (tabloul e alocat la o adresă fixă)
⇒ nu putem atribui `a` = *adresă*, dar putem atribui `pa` = *adresă*
`pa` e o *variabilă* ⇒ ocupă spaţiu de memorie şi are o adresă `&pa`
Diferenţa: `sizeof(a)==LEN*sizeof(tip)` `sizeof(pa)==sizeof(tip *)`



Tablouri şi pointeri (continuare)

Fie `char t[10];` `t` e o *adresă* (constantă) de tipul `char *`
⇒ scriem `scanf("%9s", t);` fără a fi nevoie de `&t`
`&t` are tipul `char (*)[10]` (adresă de tablou de 10 caractere)
şi aceeasi valoare ca `t` (adresa primului caracter).
Fie `char *p = t;` Corect: `scanf("%9s", p);` Greşit: `scanf("%9s", &p);`
La *adresa* `&p` (de tip `char **`) se află *valoarea* lui `p` (de tip `char *`)
`scanf` are nevoie de o adresă `char *` (`p` sau `t`) cu loc pentru 10 caractere

Tablou multidimensional = de fapt tablou unidimensional de tablouri.
`char a[6];` `a[3]` e `char` `a` e constantă `char *`
`int m[5][8];` `m[2][3]` e `int` `m[2]` e constantă `int *`
`m[2]` e un element al tabloului `m`: un tablou (linie) de 8 întregi
⇒ `sizeof m[2] == 8 * sizeof(int)`, nu `sizeof(int *)`.
⇒ `m` are tipul `int (*)[8]` (pointer la linie de 8 `int`), nu `int **`.

Atenţie! Verificaţi corespondenţa tipurilor în expresii!

Tablouri şi apeluri de funcţii

În declaraţii de funcţii, o declaraţie de parametru "*tablou de tip*" e considerată declaraţie de parametru "*pointer la tip*" ⇒ e echivalent:
`int f(int a[5]); int f(int a[8]); int f(int a[]); int f(int *a);`
⇒ nu se transmite un tablou (bloc de memorie) la funcţii, ci o adresă
⇒ nu are rost să specificăm *prima* dimensiune de tablou (`a[5]`, `a[8]`)
Parametrul adresă *nu conţine informaţii despre dimensiunea tabloului*
⇒ pt. a prelucra tabloul, funcţia are nevoie de lungime din altă sursă
(parametru la funcţie; variabilă globală; fanion de sfârşit în tablou)

```
void f(size_t l, double *t) // ca si void f(size_t l, double t[])
// size_t (stddef.h): tip pt. dimens. >= 0 (unsigned/long unsigned)
{ size_t i; for (i = 0; i < l; ++i) /* prelucreaza t[i] */ }
```

Celelalte dimensiuni trebuie indicate ⇒ tip adresă complet specificat
`int f(int t[][6]); // ca si int f(int (*t)[6]), NU int f(int *t[6])`
(var. 2: tablou de (6) `int *`, nu pointer la (tablou de) tablou de 6 `int`)

În C99, se pot specifica parametri tablou de dimensiuni variabile:
`void f(int m, int n, int a[m][n]); // sau: ..., int a[][n])`

Aritmetica cu pointeri

1. *Adunarea* unui întreg la un pointer Fie: *tip* `a[LEN];` (sau *tip* `*a;`)
`a + k` e echivalent cu `&a[k]` iar `*(a + k)` e echivalent cu `a[k]`
`a + k` e o adresă mai mare decât `a` cu dimensiunea a `k` obiecte de *tip*
⇒ nu avem aritmetică cu `void *` (nu ştim dimensiunea obiect. indicat)
⇒ adunăm nr. de *octeţi* convertind adresa la `char *` (dim. obiect =1)
Pt. *tip* `*a`, valoarea `a + k` = valoarea `(char *)a + k * sizeof(`*tip*`)`

Ex.: parcurgere de tablou; pointer echivalent cu baza + indice

```
double a[10];                    double *p;
for (i = 0; i < 10; ++i)         for (p = a; p < a + 10; ++p)
  // prelucrează a[i]              // prelucrează *p
```

2. *Diferenţa*: doar între *doi pointeri de acelaşi tip* `tip *p, *q;`
= numărul (trunchiat) de obiecte de `tip` care încap între cele 2 adrese
– diferenţa numerică în octeţi: se convertesc ambii pointeri la `char *`
`p - q == ((char *)p - (char *)q) / sizeof(tip)`

Nu sunt definite nici un fel de alte operaţii aritmetice pentru pointeri !
Se pot însă efectua operaţii logice de comparaţie (`==`, `!=`, `<`, etc.)

Pointeri şi tablouri multidimensionale

Fie declaraţia `tip a[DIM1][DIM2];` Atunci `&a[i] == a+i` = adresa liniei `i` = constantă de tipul `tip (*)[DIM2]` (adresă de tablou de `DIM2` elem. `tip`)
`a[i]` e un tablou de `DIM2` `tip`, deci ca nume de tablou are tipul `tip *`
`a[i][j]` este al `j`-lea element din linia (tabloul de `DIM2` elem.) `a[i]` şi are adresa `&a[i][j] == (tip *)(a + i) + j == (tip *)a + DIM2*i + j`

Putem parcurge un tablou cu indici sau cu pointeri. Fie `int m[5][8];`

```
int i,j;                         int (*lp)[8], *ip;
for (i=0; i<5; ++i)              for (lp=m; lp<m+5; ++lp)
  // lp e pointer la tablou de 8 int deci ++lp avanseaza cu 8 int
  // m[i] aici e echivalent cu (*lp) aici = tablou de 8 int
  for (j=0; j<8; ++j)              for (ip=*lp; ip<*(lp+1); ++ip)
  // m[i][j] aici echivalent cu (*lp)[j] echivalent cu *ip
```

`lp` şi `*lp` au valori numerice egale (adresa unei linii) dar tipuri diferite:
`lp`: pointer la tot tabloul (8 `int`); `*lp`: tabloul = pointer la elem. `int`
⇒ pentru a obţine un element, parcurgem de la `*lp` sau îl indexăm

Şiruri de caractere

tablouri de caractere cu *sfârşit* indicat convenţional de *caract. nul* '\0'
toate funcţiile standard (din `string.h`; `printf/scanf`) folosesc/cer '\0'

O constantă `"şir"` este un `char *` către memoria unde se află sirul
(în orice context, mai puţin în `sizeof` şi iniţializator de tablou).
⇒ `char s[8]; scanf("%7s", s); if (s == "txt") /* ... */`
va compara pointeri, şi nu conţinutul (în acest caz, dă sigur *fals*)

Declaraţiile a) `char s[] = "sir";` şi b) `char *s = "sir";` sunt diferite!
– a) rezervă spaţiu doar pt. şirul `"sir"+ '\0'`; *adresa* `s` e o constantă
conţinutul lui `s` poate fi modificat (în limitele dimensiunii de 4 octeţi!)
– b) rezervă spaţiu şi pentru pointerul `s`, care poate fi reatribuit
`"sir"` cf. standardului e o *constantă*, e gresşit să modificăm `s[1] = 'a';`

`char s[12][4]={"ian",...,"dec"};` şi `char *s[12]={"ian",...,"dec"};`
primul e un tablou 2-D de caractere, al doilea e un tablou de pointeri

Atenţie! `char s[3] = "sir";` nu are loc în tablou pentru '\0' final !
`char *p = "txt"; strcpy(p, "test");` sau `strcat(p, "12");` suprascrie
dincolo de memoria alocată iniţial constantei `"txt"` !

Aplicaţii: funcţii cu şiruri de caractere (`string.h`)

```
size_t strlen(const char *s) { // lungimea şirului s, până la \0
// size_t (stddef.h): tip pt. dimensiuni pozitive (unsigned sau long unsigned)
  char *p = s;       // adică char *p; p = s;
  while (*p) p++;    // avansează până întâlneşte '\0'
  return p - s;      // nr. de caract. între s şi p; '\0' nu e numărat
}
char *strcpy(char *dest, const char *src) { // copiază src în dest
  char *p = dest;
  while (*p++ = *src++);  // copiază până la '\0'; trebuie să avem loc !!!
  return dest;            // returnează dest prin convenţie
}
char *strcat(char *dest, const char *src) { // concatenează src la dest
   char *d = dest;  // trebuie sa avem loc in coada lui dest !!!
   while (*d); ++d; while (*d++ = *src++);
   return dest;
}
char *strchr(const char *s, int c) { // caută primul caracter c în s
  do if (*s == c) return s; while (*s++); // returnează pointer la car. găsit
  return NULL;                // sau NULL dacă nu a fost găsit
}
int strcmp (const char *s1, const char *s2) { // compară 2 siruri
  while (*s1 == *s2 && *s1) { s1++; s2++; } // egale dar nu '\0'
  return *s1 - *s2; // < 0 pt. s1<s2, > 0 pt. s1>s2, 0 daca egale
}
```


Funcţii cu şiruri de caractere (cont.)

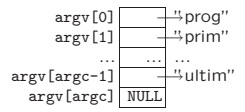
```
char *strncpy(char *dest, const char *src, size_t n) {
  char *p = dest; // copiază cel mult n caractere
  while (n-- && (*p++ = *src++));  // nu pune \0 dacă copiază fix n
  return dest;
}
int strncmp (const char *s1, const char *s2, size_t n) {
  if (n == 0) return 0;        // compară pe lungime cel mult n
  while (--n && *s1 == *s2 && *s1) { s1++; s2++; }
  return *s1 - *s2; // < 0 pt. s1<s2, > 0 pt. s1>s2, 0 daca egale
}
char *strstr(const char *where, const char *what); /* caută prima apariţie
   a şirului 1 in şirul 2; returnează pointer la locul găsit sau NULL */
size_t strspn(const char *s, const char *accept); /* câte caractere consecutive
   de la începutul lui s sunt din mulţimea de caractere din şirul accept */
size_t strcspn(const char *s, const char *reject); /* câte caractere consecutive
   de la începutul lui s NU sunt din mulţimea de caractere din şirul reject */
void *memset(void *s, int c, size_t n); // setează n octeţi cu c
void *memcpy(void *dest, const void *src, size_t n); // copiaza n octeţi
void *memmove(void *dest, const void *src, size_t n);
// copiază n octeţi; corect şi pentru zone de memorie suprapuse
```


Argumentele funcţiei `main`

Permit accesul la parametrii (argumentele) cu care programul
e rulat din linia de comandă (ex. opţiuni, nume de fişiere)
C prevede şi returnarea de program a unui cod întreg
(folosit pentru a semnala succes sau o condiţie de eroare)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;

  printf("Numele programului: %s\n", argv[0]);
  if (argc==1) printf("Nu are parametri\n");
  else for (i = 1; i < argc; i++)
    printf("Parametrul %d: %s\n", i, argv[i]);
  return 0;  /* codul returnat de program */
}
```



– dacă `argc` > 0, `argv[0]` e numele programului
– `argv[1]`, etc.: parametrii, aşa cum au fost separaţi de spaţii
– `argv[argc]` e `NULL` (marchează sfârşitul argumentelor)

Argumentele lui `main` şi conversii din şiruri

`main` poate fi definit *numai* cu parametri (`int, char *[]`), sau (`void`)
⇒ parametrii `argv[i]` sunt *întotdeauna* siruri.
Dacă şirurile reprezintă altceva, de ex. numere, trebuie *convertite*.
Conversii din şir în număr (`stdlib.h`)
1. `long strtol(const char *s, char **endptr, int base);`
– acceptă spaţii albe iniţiale; semn; consideră şirul în baza dată (2..36)
– dacă `endptr!=NULL`, primeşte adresa primului caracter neconvertit
(util pt. test de eroare, sau prelucrarea restului şirului)
Corect: `char s[9], *e; long l; l=strtol(s, &e, 10); if (*e == ...)`
Greşit: `char **e; l=strtol(s, e, 10); // e nu indică memorie validă`
Validaţi întotdeauna datele de intrare !
2. `int atoi(const char *s); // ASCII to int; == strtol(s, NULL, 10)`
– nu semnalează erori (returnează 0, care e şi o valoare validă)
1a. `double strtod(const char *s, char **endptr); // doar baza 10`
2a. `double atof(const char *s); // ASCII to floating point`
3. cu `sscanf` ⇒ se pot testa erori; `%n` pt. continuarea prelucrării

Pointeri la funcţii

Adresa unei funcţii se poate obţine, memora, şi utiliza pentru a o apela.
Sintaxa decl. funcţie: *tip_rez* `func` (*tip1*, ..., *tipn*);
Sintaxa decl. pointer funcţie: *tip_rez* `(*pf)` (*tip1*, ..., *tipn*);
⇒ atribuire `pf = func;` sau `pf = &func;` apel `pf(...);` sau `(*pf)(...);`
(*numele* funcţiei e echivalat cu *pointerul* la funcţie)

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

Mai jos: definim unui tablou de pointeri de funcţii (ex. pt. un meniu)
Pentru claritate, declarăm un tip: `typedef void (*pfun_t)(void);`

```
void help(void); void menu(void); /*...*/ void quit(void);
pfun_t funtab[10] = { help, menu, ..., quit };
int k = getchar() - '0'; if (k >= 0 && k <= 9) funtab[k]();
```

Parametri pointeri la funcţie

Utilizaţi pentru a parametriza funcţii generice de prelucrare
Algoritmul *quicksort*, declarat (în `stdlib.h`) ca funcţie cu parametrii:

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

– adresa tabloului de sortat, numărul şi dimensiunea elementelor
– adresa funcţiei care compară 2 elemente (returnează $<$, $=$ sau $>$ 0)
(e implementată de programator în funcţie de tipul ce trebuie sortat)
– foloseşte argumente `void *` fiind compatibile cu pointeri la orice tip

în scrierea unor astfel de funcţii:
– forţăm `(void *)` la `(char *)` pt. aritmetică (deplasamente de octeţi)
în scrierea funcţiei date ca parametru:
– forţăm param. `void *` la tipul actual (cunoscut la scrierea funcţiei)
ex. `int intcmp(const void *p, const void *q) { return *(int *)p - *(int *)q; }`
– sau se scrie funcţia cu tipul actual, şi se forţează tipul funcţiei la apel
ex. `int intcmp(int *p, int *q) { return *p - *q; }` şi apelul
`qsort(..., (int (*)(const void *, const void *))intcmp);`

Alocarea dinamică

pt. gestionarea memoriei după nevoile ce apar la *rularea* programului
`void *calloc(size_t num, size_t size);` toate declarate în `stdlib.h`
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 returnează adresa alocată sau `NULL` la 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 şi cum folosim alocarea dinamică ?

NU e necesară când ştim 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);
```

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);
  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; // initializare pentru realloc
  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))) {
        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;
}
```


Exemplu: alocare dinamică + sortare

```
#include <stdio.h>
#include <stdlib.h>
#define INCR 100 // alocăm pt. 100 de numere odată
typedef int (*cmpptr)(const void *, const void *); // tip pt. qsort
int cmp(int *p, int *q) { return *p - *q; } // cu tipul concret int
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), (cmpptr)cmp); // trebuie typecast la cmp
  for (n = 0; n < i; n++) printf("%d ", t[n]);
  free(t); // nu uitam sa eliberam memoria !
  return 0;
}
```