Pointeri și tablouri. Aritmetica cu pointeri

2 mai 2006

Pointeri: recapitulare

– o variabilă `x` de tipul *tip* are o adresă `&x` de tipul *tip* `*` (pointer la *tip*)
– numele unui tablou e adresa primului element
  dacă elementele au tipul *tip*, numele tabloului are tipul *tip* `*`
  elementele pot avea tip tablou (celelalte dimensiuni mai puțin prima)
– un parametru tablou la o funcție *tip-rez* `f(`*tip* `tab[])`
  e de fapt un parametru pointer la *tip*: *tip-rez* `f(`*tip* `*tab)`
– având adresa unui obiect, putem folosi dar și modifica valoarea sa
  (ex.: funcție cu parametru tablou poate folosi/modifica elementele)

Discutăm în continuare:
– Cum folosim (ne referim la) un obiect (variabilă) știind adresa sa ?
– Pentru ce folosim adresele (ex. ca parametri), în afară de tablouri ?
– Care e relația dintre o adresă de element și un indice (la tablouri) ?

Tipuri pointer. Declarare. Indirectare

Dacă variabila `x` are tipul `tip` atunci adresa `&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 (și parametri) de tip adresă (pointeri):
tip `*` *nume_var*; *nume_var* e pointer la (adresă pt.) o valoare de *tip*
pointer = o variabilă a cărei valoare e o *adresă* (a altei variabile)

Dacă `p` e o adresă, atunci `*p` reprezintă obiectul de la adresa `p`
Operatorul prefix `*` e operatorul de *indirectare* (dereferențiere, referire indirectă prin adresă). Poate fi aplicat doar unei adrese (pointer).
Dacă pointerul `p` are tipul *tip* `*` atunci obiectul `*p` are tipul *tip*

Sintaxa declarației (aceeași dar citită în două feluri) sugerează folosirea:
`char* p;` `p` e o variabilă de tipul `char *` (adresă de `char`)
`char *p;` (scris uzual): `*p` (obiectul de la adresa `p`) are tip `char`
Atenție: `int *p, *q;` : doi pointeri; `int *p, q;` : un pointer, un întreg

Operatorii de adresă și dereferențiere

Pointerii au adrese, ca orice variabile:
pt. `int *p;` adresa `&p` are tipul `int **`
`int ** pp = &p;` ⇒ `pp` are tipul `int **`,
adică adresa unei adrese de `int`
putem grupa: `int* *pp` sau `int **pp`
deci `*pp` are tipul `int *` (adresă de `int`)
și `**pp` are tipul `int` (val. de la adr. `*pp`)

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

Ca orice variabilă, unui pointer trebuie dată o valoare înainte de folosire
de ex. adresa unei variabile de acel tip: `int x, *p, **pp; p=&x; pp=&p;`

O *referință* `*p` se folosește ca o variabilă, *la stânga sau la dreapta unei atribuiri* (de ex. dacă `p = &x` atunci `*p` se folosește absolut la fel ca `x`)
`int x, y, z, *p; p = &x; z = *p; /* z = x */ *p = y; /* x = y */`

OBS: Operatorii adresă `&` și de indirectare `*` sunt *unul inversul celuilalt*:
  `*&x` este chiar `x`, pentru orice obiect (variabilă) `x`
  `&*p` are valoarea `p`, pentru orice variabilă pointer `p`
Adresa unui obiect (`&x`, `&*p`) nu poate fi atribuită (modificată).

Pointeri ca argumente/rezultate de funcții

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 *conținutul* ei
  ex. `scanf`: primește *adrese*, completează conținutul cu valorile citite

```
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
}
```

Obs: în funcție s-a lucrat cu *conținutul* de la adresele pa și pb
Ex.: `int x = 3, y = 5; swap(&x, &y); // acum x = 5 şi y = 3`
OBS: Nu se poate obține acest efect cu `void swap(int m, int n);`
parametrii se transmit *prin valoare* (indiferent de unde provine ea)

Pointeri ca argumente de funcții (cont.)

– dacă funcția trebuie să producă mai multe valori

```
void minmax(double a[], size_t len, double *pmin, double *pmax) {
  *pmin = *pmax = a[0];
  for (int i = 1; i < len; ++i) // C99
    if (a[i] < *pmin) *pmin = a[i];
    else if (a[i] > *pmax) *pmax = a[i];
}
double a[3] = { .1, 6.7, -2.3}; double m, M; minmax(a, 3, &m, &M);
```

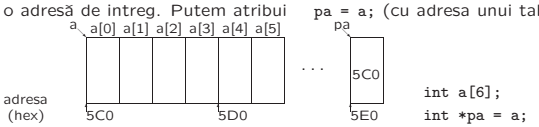
  sau `scanf`: citește valori la adresele argument, returnează câte a citit
– dacă trebuie sa returnăm o valoare, dar sunt cazuri de excepție

```
int rad2(double a, double b, double c, double *px1, double *px2) {
  double delta = b*b - 4*a*c;
  if (delta < 0) return 0; // nu avem soluţii
  *px1 = (-b - sqrt(delta))/(2*a); *px2 = (-b + sqrt(delta))/(2*a);
  return 1 + (delta > 0); // nr. de soluţii: 1 sau 2
}
double x1, x2; int nrsol = rad2(1, 4, 2.5, &x1, &x2);
```

Tablouri şi pointeri

Declararea unui tablou alocă un bloc de memorie pt. elementele sale
⇒ *numele tabloului e adresa* blocului respectiv (= a primului element)
⇒ 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)

Declarând `tip *pa;` nu alocăm loc pentru nici un întreg, doar pentru o adresă de intreg. Putem atribui `pa = a;` (cu adresa unui tablou).




```
int a[6];
int *pa = a;
```

Diferenţe: adresa `a` e o *constantă* (tabloul e alocat la o adresă fixă)
⇒ putem modifica conţinutul (`a[i]`) dar nu putem atribui `a` = *adresă*
`pa` e o *variabilă* ⇒ ocupă memorie şi o putem atribui, de ex. `pa = a`
`sizeof a == 6*sizeof(int)` diferit de `sizeof pa == sizeof(int *)`

Pointeri şi indici

Termenul "pointer" provine de la "a indica" – to point (to).
Un element de tablou `a[i]` e identificat prin numele tabloului `a`
şi indicele `i`. Mai simplu: printr-o singură adresă (pointer): `&a[i]`

```
char *strchr(const char *s, int c) { // caută caracter în şir
  for (int i = 0; s[i]; ++i)          // C99
    if (s[i] == c) return &s[i];      // găsit: returnează adresa
  return NULL; // s-a ajuns la sfârsit: returnează ‘‘invalid’’
}
```

`NULL` : valoare care indică o adresă invalidă: `(void *)0` , din `stddef.h`

Adresa unui pointer poate fi transmisă parametru la o funcţie, care poate astfel să-i modifice *valoarea* (ca pt. orice variabila):
Ex.: functie care converteşte din şir în întreg, şi indică unde s-a oprit:

```
long strtol(const char *nptr, char **endptr, int base); // stdlib.h
char s[] = "  -123xy"; char *end; long n;
n = strtol(s, &end, 10);
printf("n este %ld, a rămas din sir %s\n", n, end);
```


Aritmetica cu pointeri

O variabilă `v` de un anumit `tip` ocupă `sizeof(tip)` octeţi
⇒ `&v + 1` reprezintă adresa la care s-ar putea memora următoarea variabilă de acelaşi tip (adresa cu `sizeof(tip)` mai mare decât `&v`).

1. *Adunarea* unui întreg la un pointer: poate fi parcurs un tablou
`a + i` e echivalent cu `&a[i]` iar `*(a + i)` e echivalent cu `a[i]`

```
char *strchr(const char *s, int c) { // caută c în s
  while (*s)
    if (*s == c) return s; // gata, găsit
    else ++s; // altfel avansează
  return NULL; // n-a fost găsit
}
```

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];` Elementul `a[i][j]` este al `j`-lea element din tabloul de `DIM2` elemente `a[i]` şi are adresa
`&a[i][j] == (tip *)(a + i) + j == (tip *)a + DIM2*i + j`

⇒ pentru compilarea expresiei `a[i][j]` e necesară cunoaşterea lui `DIM2`
⇒ în declaraţia unei funcţii cu parametri tablou trebuie precizate toate dimensiunile în afară de prima (irelevantă): `void f(int m[][5]);`

Pointeri şi şiruri

Declaraţiile `char s[] = "sir";` şi `char *s = "sir";` sunt diferite!
– prima rezervă spaţiu doar pt. şirul `"sir"`, iar adresa `s` e o constantă
– a doua rezervă spaţiu şi pentru pointerul `s`, care poate fi reatribuit

`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

Argumentele liniei de comandă

Limbajul C permite accesul la parametrii argumentele) cu care programul e rulat din linia de comandă (ex. opţiuni, nume de fişiere)
De asemenea, permite returnarea de program a unui cod întreg (folosit uzual 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("Program apelat fără parametri\n");
  else for (i = 1; i < argc; i++)
    printf("Parametrul %d: %s\n", i, argv[i]);
  return 0;  /* codul returnat de program */
}
```

– `argv[0]` e numele programului, deci întotdeauna `argc >= 1`
– `argv[1]`, etc.: parametrii, aşa cum au fost separaţi de spaţii

Eroarea cea mai frecventă: absenţa iniţializării

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

Pointerii trebuie iniţializaţi, 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)

`NULL` definit în `stddef.h` ca `(void *)0`: nu e o adresă validă
⇒ folosit (la iniţializări, sau returnat) ca valoare de pointer invalid

OBS: pointerii au valori numerice, dar nu sunt acelaşi lucru ca întregii.
⇒ Nu convertiţi între pointer şi `int` (e dependent de implementare).

OBS: Un prim test al corectitudinii programului: *verificarea de tipuri*
Verificaţi că expresiile au tipuri corespunzătoare (ex. la atribuire)
⇒ valabil şi pentru pointeri (nu confundaţi `p` cu `*p`, etc.)