Limbaje de programare

Pointeri. Alocare dinamică

24 noiembrie 2009

Pointeri: recapitulare

O variabilă `x` de tipul *tip* are o *adresă* `&x` de tipul *tip* `*`

Adresele sunt nenule. Valoarea `NULL` (adresă 0) indică o adresă invalidă.

Variabila `x` ocupă `sizeof(x)` (sau: `sizeof(`*tip* `))` octeți pornind de la `&x`

Adrese și tablouri

Numele `t` al tabloului *tip* `t[5];` e *adresa* tabloului
(a primului element, `&t[0]`). Adresa `t` are tipul *tip* `*`

Funcțiile au ca parametri *adresa* tabloului, NU conținutul său

`void f(`*tip* `t[8]);` e la fel ca `void f(`*tip* `t[])` și ca `void f(`*tip* `*t);`

Funcția care primește adresa unei variabile o poate *modifica* (și citi).

Ex: `scanf` (atribuie valori citite de la intrare), funcții cu tablouri
(modifică *conținutul* tabloului, dar nu *adresa*, transmisă prin *valoare*!)

Pointeri: recapitulare

Șiruri de caractere

O *constantă șir de caractere* `"sir"` are tipul `char *`
Valoarea constantei `"sir"` este *adresa* de memorie unde se află șirul.

ATENȚIE Nu putem compara un `char` (`'a'`) cu un șir (adresă) `"a"` !
Comparăm șiruri cu `str(n)cmp`, NU cu `==`
Operatorul `==` compară *adrese* (UNDE se află șirul), NU conținutul (CE șir este)

Variabile pointer

Pointerii se folosesc ca și orice alte variabile
– au tip, valoare, loc în memorie, adresă
– pot fi declarați, atribuiți, tipăriți, dați parametri
– au operații specifice (dereferențiere `*`, aritmetică `+ - ++ --`)

Declararea pointerilor. Adrese

Pointer = o variabilă care conține *adresa* altei variabile

Declararea pointerilor

tip **nume_var*; // *nume_var* `e pointer la o valoare de` *tip*

Operatorul adresă `&`

operator prefix

– operand: o variabilă (ex. `x`); rezultat: *adresa* variabilei `&x`

– folosit doar pt. *variabile* (și elem. tablou), nu constante, expresii, etc.

– se poate atribui unui pointer la acel tip: `int x; int *p; p = &x;`

Ce valoare se află la o adresă?

Operatorul de dereferențiere (indirectare) * operator prefix

– operand: pointer; rezultat: *obiectul* (variabila) indicat de pointer

– `*p` e un *lvalue*, poate fi folosit la stânga unei atribuiri, ca și variabilele sau elem. tablou; (orice *expresie* poate fi la dreapta lui `=`)

– dacă `p` e `&x`, atunci `*p` e obiectul de la adresa `p` (a lui `x`), deci `x`

```
int x, y, *p; p = &x;      y = *p;  /* y = x */      *p = y; // x = y
```

Operatorul * e *inversul* lui `&`: `*&x` e chiar `x` (obiectul de la adresa lui x)

`&*p` e `p` (`p`: pointer cu valoare validă): adresa obiectului de la adresa `p`

Declarație și dereferențiere

Putem citi declarația *tip* ∗ *p*;

tip ∗ `p;` `p` are tipul *tip* ∗

tip `*p;` `*p` e un caracter

```
char **s; // adresă de adr.de char

char *t[8]; // tab.de 8 adr.de char
```

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

ATENȚIE O *declarație* cu *inițializare* NU este o *atribuire* !

`int t[2] = { 3, 5 };` inițializează `t`. NU are sens: ~~`t[2] = { 3, 5 };`~~

`int x, *p = &x;` este `int x; int *p = &x;` sau `int x; int *p; p = &x;`
(e inițializat/atribuit `p`, NU `*p`). ~~`*p = &x`~~ e incorect ca tip!

`char *p = "sir";` e `char *p; p = "sir";` dar ~~`*p = "sir;"`~~ e greșit!

EROARE: lipsa inițializării

E o *EROARE* să folosim o *variabilă neinițializată*

```
{ int sum; for (i=0; i++ < 10; ) sum += a[i]; } // cât e iniţial ???
```

⇒ programul începe calculul cu o valoare *la întâmplare !!*

Pointerii, ca orice variabile trebuie inițializați!

– cu *adresa* unei variabile (sau cu alt pointer inițializat deja)

– cu o adresă de memorie *alocată dinamic* (vom discuta ulterior)

EROARE: *tip* `*p; *p =` *ceva*; *EROARE*: `char *p; scanf("%s", p);`

– p este *neinițializat* (eventual nul, dacă e variabilă globală)

⇒ valoarea va fi scrisă la o *adresă de memorie necunoscută* (evtl. nulă)

⇒ memorie coruptă, vulnerabilități de securitate, rulare abandonată

ATENȚIE: un pointer nu este un întreg. Greșit: ~~`int *p = 640;`~~ !

NU putem alege adresa unei variabile (unde să fie dispusă în memorie)

⇒ se determină la încărcarea programului / când se alocă memoria

Pointeri ca argumente/rezultate de funcții

Având adresa `p` a unei variabile îi putem *modifica valoarea*: `*p = ...`
funcția care primește adresa unei variabile poate modifica valoarea ei
ex. `scanf` primețe *adrese*, completează *conținutul* cu valorile citite
dar parametrii sunt transmiși *tot prin valoare*: adresa nu se modifică

```
void swap (int *pa, int *pb) {  // schimba valorile de la 2 adrese
  int tmp;  // variabila temporara pentru valoarea schimbata prima
  tmp = *pa; *pa = *pb; *pb = tmp;   // trei atribuiri de intregi
}
```

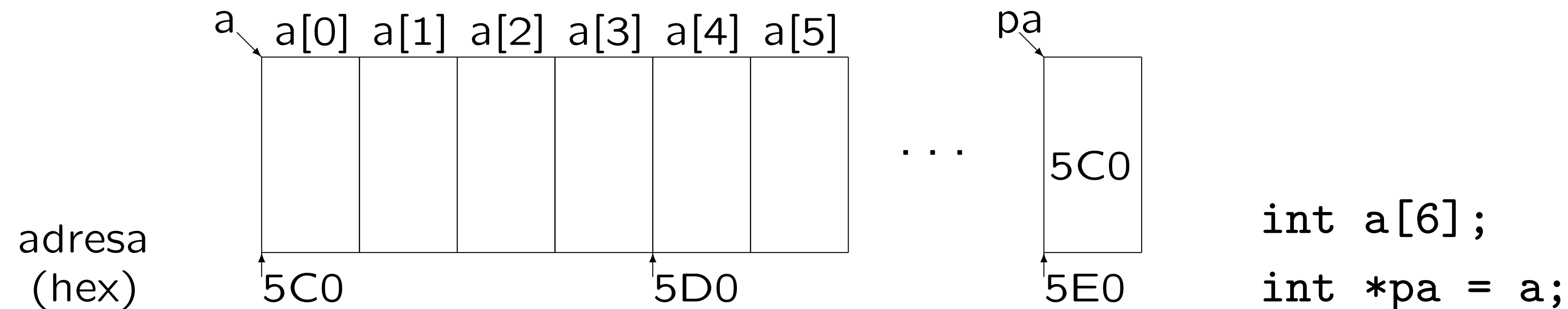
Ex.: `int x = 3, y = 5; swap(&x, &y); // acum x = 5 şi y = 3`

Folosim adrese ca parametri de funcții:
– ca să transmitem tablouri (altfel nu se poate)
– pentru a întoarce mai multe rezultate (funcția permite doar unul)
ex. minimul *și* maximul unui tablou; rezultat *și* cod de eroare

Tablouri și pointeri

În limbajul C noțiunile de *pointer* și *nume de tablou* sunt *asemănătoare*.
– declararea unui tablou alocă un bloc de memorie pt. elementele sale
– *numele* tabloului e *adresa* blocului respectiv (= a primului element)
Dacă declarăm *tip* `a[LEN], *pa;` putem atribui `pa = a;`
`&a[0]` e echivalent cu `a` iar `a[0]` e echivalent cu `*a`

Diferența: 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 spațiu de memorie și are o adresă `&pa`




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

Tablouri și pointeri (continuare)

Ca parametri la funcții, cele două scrieri înseamnă *același lucru*

`size_t strlen(char s[]);` sau `size_t strlen(char *s);`

Ca declarații, de tablou / pointer, *sunt diferite*!

Tablou: `char s[] = "test";` `s[0]` e `'t'`, `s[4]` e `'\0'` etc.

`s` e o *adresă constantă* de tip `char *`, nu variabilă cu loc în memorie

NU se poate atribui `s = ...`, se poate atribui `s[0] = 'f'`

`sizeof(s)` e `5 * sizeof(char)` `&s` e chiar `s`

(dar are alt tip, adresă de tablou de 5 char: `char (*)[5]`)

Pointer: `char *p = "test";` `p[0]` e `'t'`, `p[4]` e `'\0'` etc. (la fel)

`p` e o *variabilă de tip adresă* (`char *`), ocupă loc în memorie

NU se poate atribui `p[0] = 'f'` ("test" e o constantă șir),

se poate atribui `p = "ana";` sau `p = s;` și apoi `p[0] = 'f'`

`sizeof(p)` e `sizeof(char *)` `&p` NU e `p`

⇒ e GREȘIT: `scanf("%4s", &p);` CORECT: `scanf("%4s", p);`

Aritmetica cu pointeri

O variabilă `v` de un anumit `tip` ocupă `sizeof(tip)` octeți
⇒ `&v + 1` reprezintă adresa la care s-ar putea memora următoarea variabilă de același 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 *endptr(char *s) { /* returnează pointer la sfârşitul lui s */
  char *p = s;          /* sau: char *p; p = s; */
  while (*p) p++;       /* adică la poziţia marcată cu '\0' */
  return p;
}
```

2. *Diferența*: doar între doi pointeri *de același tip* `tip *p, *q;`
= numărul (trunchiat) de obiecte de `tip` care încap între cele 2 adrese
– diferența numerică în octeți: se convertesc ambii pointeri la `char *`

```
p - q == ((char *)p - (char *)q) / sizeof(tip)
```

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

Pointeri și indici

Termenul "pointer" provine de la "to point (to)" (a indica)

Când identificăm un element de tablou `a[i]` folosim doua variabile: tabloul și indicele, și implicit o adunare (indicele la adresa de bază)

Mai simplu: folosind direct un pointer la adresa elementului `&a[i]==a+i`
$\Rightarrow$ la parcurgere, în loc să avansăm indicele, incrementăm pointerul

```
char *strchr_i(const char *s, int c) {  // caută caracter în şir
  for (int i = 0; s[i]; ++i) // parcurge s cu indice i până la '\0'
    if (s[i] == c) return &s[i]; // s-a găsit: returnează adreasa
  return NULL; // nu s-a găsit: returnează NULL (adresă invalidă)
}

char *strchr_p(const char *s, int c) {  // scrisă folosind pointer
  for ( ;*s; ++s)     // folosim chiar parametrul pentru parcurgere
    if (*s == c) return s;        // s indică caracterul curent
  return NULL;        // nu s-a găsit
}
```

Pointeri și tablouri multidimensionale

Fie un tablou bidimensional (matrice) declarat *tip* `a[DIM1][DIM2];`
`a[i]` e adresa (constantă *tip* `*`) a unui tablou (linii) de `DIM2` elemente
`a[i][j]` e al `j`-lea element din tabloul de `DIM2` elemente `a[i]` ; adresa
`&a[i][j] == a[i]+j` e cu `DIM2*i+j` elemente după adresa tabloului `a`
⇒ o funcție cu parametri tablou trebuie să cunoască toate dimensiunile
în afară de prima ⇒ trebuie declarată *tip-f* `f(`*tip-t* `t[][DIM2]);`

`char t[12][4]={"ian",...,"dec"};` și `char *p[12]={"ian",...,"dec"};`

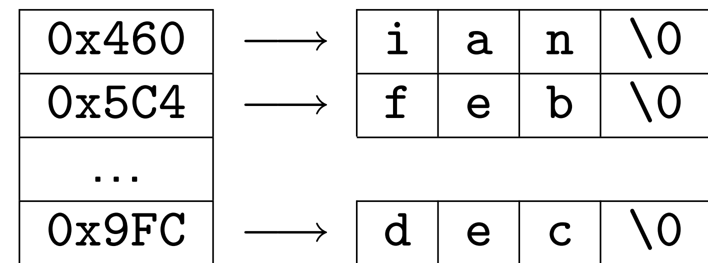
`t` e un tablou 2-D de caractere

i	a	n	\0
f	e	b	\0
...			
d	e	c	\0

`t` ocupă 12 * 4 octeți

`t[6] = ...` e GREȘIT
(`t[6]` e adresa constantă a liniei 7)

`p` e un tablou de pointeri



`p` ocupă 12*sizeof(char *) octeți
(+ 12*4 octeți pt. *constantele* șir)

`p[6]="iulie"` modifică o adresă
(elementul 7 din tabloul de adrese `p`)

Argumentele liniei de comandă

Pe linia de comandă, după *numele programului*, pot urma *argumente* (parametri): opțiuni, nume de fișiere ... Exemple:

`gcc -Wall -o prog prog.c` `ls` *director* `cp` *fisier1 fisier2*

În C, avem acces la linia de comandă declarând `main` cu 2 parametri:

`int argc` : nr. de cuvinte din linia de comandă (nr. argumente + 1)

`char *argv[]` : tablou cu adresele argumentelor (șiruri de caractere)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Numele programului: %s\n", argv[0]);
  if (argc == 1) printf("Program apelat fără parametri\n");
  else for (int i = 1; i < argc; i++)
    printf("Parametrul %d: %s\n", i, argv[i]);
  return 0;  /* codul returnat de program */
}
```

`argv[0]` (primul cuvânt) e numele programului, deci sigur `argc >= 1`

tabloul `argv[]` e încheiat cu un element `NULL` (`argv[argc]`)

Funcții de citire/scriere formatată în șiruri

Funcțiile de tipul `printf/scanf` pot avea ca sursă/dest. și șiruri de char.

```
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

Pentru `sprintf`, poate apărea problema depășirii tabloului în care se scrie, dacă acesta nu e dimensionat corect (suficient). Se recomandă:

```
int snprintf(char *str, size_t size, const char *format, ...);
```

în care scrierea e limitată la `size` caractere ⇒ variantă sigură

Între funcții similare, trebuie alese cele corespunzătoare situației. Ex:

```
int n, r; char *s, *end;
n = atoi(s); // doar când s e bun; nu semnalează erori
n = strtol(s, &end, 10); /* se pot testa erori (s == end) şi
                            prelucra mai departe de la end */
r = sscanf(s, "%d", &n); /* se pot testa erori (r != 1)
dar punctul de oprire în s nu e explicit (eventual cu %n) */
```

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 înseamă 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

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 marimea 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]);
```

Exemplu: citirea unei linii de dimensiune nelimitată

```
#include <stdio.h>
#include <stdlib.h>
#define BLOCK  16
char *getline(void) {
  char *p, *s = NULL; // s initializat pentru realloc
  int c, lim = -1, size = 0; // pastram un loc pentru \0
  while ((c = getchar()) != EOF) {
    if (size >= lim) // s-a umplut zona alocata
      if (!(p = realloc(s, (lim+=BLOCK)+1))) { // mai aloca 16
        ungetc(c, stdin); break; // termina daca nu mai e loc
      } else s = p;  // tine minte noua adresa alocata
    s[size++] = c;   // adauga ultimul caracter
    if (c == '\n') break; // iese la linie noua
  } // termina cu \0, realoca doar cat e nevoie
  if (s) { s[size++] = '\0'; s = realloc(s, size); }
  return s;
}
```

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.
Declarații: 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 funcţie void
funptr funtab[10]; // tablou de pointeri de funcţie 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 (*compar)(void *, void *));
```

– adresa tabloului de sortat, numărul și dimensiunea elementelor
– adresa funcției care compară 2 elemente (returnează $<$, $=$ sau $>$ 0)
$\Rightarrow$ folosește argumente `void *` fiind compatibile cu pointeri la orice tip

```
typedef int (*comp_t)(const void *, const void *);//tip ptr.fct.cmp
int intcmp(int *p1, int *p2) { return *p1 - *p2; }//fct.cmp.intregi
int tab[5] = { -6, 3, 2, -4, 0 };  // tabloul de sortat
qsort(tab, 5, sizeof(int), (comp_t)intcmp); // sorteaza crescator
```