

Pointeri – laborator 3

Variabile și 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: orice poate folosit pe partea stângă a unei atribuirii (variabilă, element de tablou, funcție; NU pentru expresii oarecare)

O adresă poate fi tipărită (în hexazecimal) cu formatul **%p** în **printf**

```
#include <stdio.h>
double d; int a[10]; /* variabile globale */
int main(void)
{
    int k; /* variabilă locală */
    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 */
} /* Obs &a[5] - &a[0] == 5 * sizeof(int) (poziții consecutive) */
```

Tipuri pointer. Declarare. Indirectare

Orice expresie în C are un tip => la fel și 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 conține adresa altei variabile

Operatorul prefix * dă obiectul ***p** de la adresa dată de operandul **p**

Operand: **pointer**. Rezultat: **referință** la obiectul indicat de pointer

=> **operator de indirectare** (dereferențiere, referire indirectă prin adresă)

OBS: Dacă pointerul **p** are tipul **tip ***, ***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; ***p** (obiectul de la adresa **p**) are tipul char

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**
dar putem citi **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

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

O **referință** ***p** poate fi folosită **la stânga sau la dreapta unei atribuiri** (în cazul de mai sus, ***p** se folosește absolut la fel (sinonim) cu **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**

(dar **p** e o variabilă și poate fi atribuită; **&*p** e o expresie și nu poate)

Eroarea cea mai frecventă: absentă inițializării

Utilizarea oricărei variabile neinițializate e o eroare logică în program !

{ int x; printf("%d", x); } /* cât e x ?? valoare la întâmplare! */

Pointerii trebuie inițializați, ca orice variabile !

- cu adresa unei variabile (sau cu alt pointer inițializat deja)
- cu o adresă de memorie alocată dinamic (vom discuta ulterior)

EROARE: tip *p; *p = valoare; p este neinițializat!! (eventual nul)

- valoarea va fi scrisă la o adresă de memorie necunoscută (evtl. nulă)
- coruperea memoriei, rezultate eronate sau imprevizibile, terminarea forțată a programului (sub sisteme de operare cu memorie protejată)

NULL definit în **stddef.h** ca **(void *)0**: nu e o adresă validă

- folosit (la inițializări, sau returnat) ca valoare de pointer invalid

OBS: pointerii au valori numerice, dar nu sunt același lucru ca întregii.

- Nu converțiți între pointer și **int** (e dependent de implementare). **OBS:** Un prim test al corectitudinii programului: **verificarea de tipuri**

Verificați că expresiile au tipuri corespunzătoare (ex. la atribuire)

- valabil și pentru pointeri (nu confundați **p** cu ***p**, etc.)

Pointeri ca argumente/rezultate de funcții

Permit **modificarea valorii unei variabile** prin transmiterea adresei ei

- variabilă se poate modifica prin indirectarea unui pointer către ea
- nu se modifică *adresa* (transmisă tot prin valoare) ci *conținutul* ei

```
void swap (int *pa, int *pb) /* schimbă val. de la adr. pa și pb */  
{  
    int tmp; /* variabilă auxiliară necesară pentru interschimbare */  
    tmp = *pa; *pa = *pb; *pb = tmp; /* trei atribuiri de întregi */  
} /* 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 și y = 3 */

OBS: Nu se poate obține efectul cu `void swap(int m, int n);`

(ar schimba valorile transmise în corpul funcției, fără efect în afară)

Folosire: c_ and limbajul nu permite transmiterea prin valoare (**tablouri**)

sau ar fi inefficientă (structuri mari) => transmem *adresa* variabilei

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)
- 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)

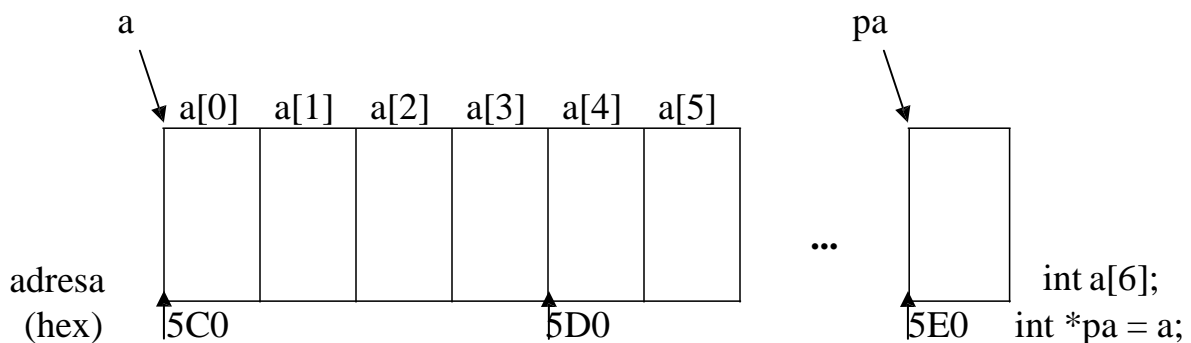
a[0] e echivalent cu ***a** (obiectul de la adresa a e primul element)

Dacă declarăm **tip *pa;** putem atribui **pa = 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**



În declarații de funcții, se pot folosi oricare din variante:

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

(de fapt, compilatorul convertește prima variantă în a doua)

`size_t`: tip pt. dimensiuni pozitive din `stddef.h` (ca și `unsigned` sau `unsigned long`)

- nu se transmit tablouri (bloc de memorie) la funcții, ci adresele lor

Fie `char t[21];` Compilatorul consideră `&t` ca fiind `t`

- s-ar putea scrie și `scanf("%20s", &t)` în loc de `scanf("%20s", t)` se recomandă totuși prima variantă, pentru uniformitate cu cazul: `char *p; p = s;`
`scanf("%20s", p) /* aici e incorect &p ! */`

Diferență între tablouri și pointeri:

`sizeof t == 21*(sizeof char)` diferit de `sizeof p == sizeof(char *)`

Atenție! Verificați corespondența tipurilor în expresii.

Ex. `char m[5][80]; char *p; p` și `m` nu au același tip, dar `p` și `m[2]` au !

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 sfârșitul lui s */
    char *p = s;                  /* sau: char *p; p = s; */
    while (*p) p++;               /* adică la poziția 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

- diferență 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.)

Aplicații: funcții cu șiruri de caractere (`string.h`)

`size_t strlen(const char *s) /*lungimea șirului s */`

`char *p = s;`

`while (*p) p++;` `/* până întâlnește '\0' */`

`return p - s;` `/* '\0' nu e numărat */`

`}`

```

char *strcpy(char *dest, const char *src) { /* dest <- src */
    char *p = dest;
    while (*p++ = *src++); /* copiază până întâlnește '\0' */
    return dest;          /* returnează dest prin convenție */
}

char *strcat(char *dest, const char *src) /* concat. src la dest */
{
    return strcpy(dest + strlen(dest), src);
}

int strcmp (const char *s1, const char *s2) { /* compară */
    while (*s1 == *s2 && *s1) { s1++; s2++; } /* egale dar nu '\0' */
    return *s1 - *s2; /* < 0 pt. s1<s2, > 0 pt. s1>s2, 0 pt. egal */
}

char *strncpy(char *dest, const char *src, size_t n) {
    char *p = dest; /* copiază cel mult n caractere */
    while (n-- && *p++ = *src++);
    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 pt. egal */
}

char *strchr(const char *s, int c) { /* caută primul c în s */
    do if (*s == c) return s; while (*s++);
    return NULL; /* dacă nu a fost găsit */
}

void *memset(void *s, int c, size_t n); /* setează n octeți cu c */
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
/* copiază n octeți; ultima variantă și pentru zone suprapuse */

```

Pointeri și tablouri multidimensionale

Fie declarația ***tip a[DIM1][DIM2];*** Elementul ***a[i][j]*** este al j-lea element din tabloul de ***DIM2*** elemente ***a[i]*** și are adresa

&a[i][j] == (tip *) (a + i) + j == (tip *) a + DIM2 * i + j

- pentru compilarea expresiei ***a[i][j]*** e necesară cunoașterea lui ***DIM2***
- în declarația unei funcții cu parametri tablou trebuie precizate toate dimensiunile în afară de prima (irelevantă): ***void f(int m[][5]);***

Pointeri și șiruri

Declarațiile ***char s[] = "sir";*** și ***char *s = "sir";*** sunt diferite!

- prima rezervă spațiu doar pt. șirul "sir", iar adresa ***s*** e o constantă
 - a doua rezervă spațiu și pentru pointerul ***s***, care poate fi reatribuit
- char s[12][4]={"ian",...,"dec"};*** și ***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. opțiuni, nume de fișiere)

De asemenea, permite returnarea de program a unui cod întreg (folosit uzual pentru a semnală succes sau o condiție 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 fara 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, așa cum au fost separați de spații

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)

Atenție la sintaxă:

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

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

Exemplu de utilizare: parametrizarea unei alte funcții

Algoritmul quicksort, declarat (în **stdio.h**) ca funcție cu parametrii:

- 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

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

- folosește argumente ***void **** fiind compatibile cu pointeri la orice tip
- pentru tabele de rutine, apelate în funcție de un indice
- exemplu: meniu cu apelare de funcții în funcție de tasta apăsată

void help(void); void menu(void); /*...*/ void quit(void);

void (*funtab)[10](void) = { help, menu, ..., quit };

void do_cmd(void)

```
{  
    int k = getchar() - '0';  
    if (k >= 0 && k <= 9) funtab[k]();  
}
```

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

typedef void (*funptr)(void); /* pointer la funcție void */

funptr funtab[10]; /* tabloul de pointeri de funcție */

Alocarea dinamică

Până acum am atribuit la pointeri doar adrese de variabile existente și am declarat static doar variabile de dimensiuni cunoscute la compilare.

Discutăm: funcții de gestiune dinamică a memoriei (**stdlib.h**):
alocarea memoriei după necesități stabilite la rularea programului

```
void *malloc(size_t size); /* alocă size octeți */  
void *calloc(size_t num, size_t size); /* num*size oct. init. 0 */  
/* m/calloc returnează NULL la eroare (ex. mem. insuficientă) */  
void *realloc(void *ptr, size_t size); /* modifică dimensiunea, poate muta  
blocul, dar păstrează conținutul memoriei */  
void free(void *ptr); /* eliberează mem. 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]);
```

Exemple de alocare dinamică

Să se citească un șir de numere, terminat cu zero și să se sorteze.

```
#include <stdio.h>  
#include <stdlib.h>  
#define NUM 100 /* alocăm pt. 100 de numere odată */  
typedef int (*cmpptr) (const void *, const void *);  
int cmp(int *p, int *q) { return *p - *q; } /* pt. sortare */  
void main(void) {  
    int i = 0, n = 0, *t = NULL; /* contor, total, tablou */  
    do { /* alocă câte NUM întregi, inițial și când e nevoie */  
        if (i == n) { n += NUM; t = realloc(t, n*sizeof(int)); }  
        scanf("%d", &t[i]); /* realloc(NULL,sz) e ca și malloc(sz) */  
    } while (t[i++] > 0);  
    qsort(t, i, sizeof(int), (cmpptr)cmp); /* sortează */  
    for (n = 0; n < i; n++) printf("%d ", t[n]);  
    free(t);  
}
```


Probleme

1. Scrieți un program care verifică dacă un vector este ordonat crescător până în poziția k și descrescător până la sfârșit.
2. Scrieți o funcție care lucrează cu pointeri și care inversează un șir de caractere.
3. Scrieți o funcție `copiază_n(char *s, char *t, n)` care copiază primele n elemente ale șirului de caractere t la sfârșitul șirului de caractere s .
4. Dându-se un tablou bidimensional cu elemente reale, ale cărei linii și coloane nu depășesc numărul 20, să se afișeze tabloul după ce fiecare element a fost înlocuit cu câtul dintre valoarea lui anterioară și valoarea minimă din tablou.
5. Se citește de la tastatură un cuvânt de lungime cel mult 20 de caractere, format numai din litere mari. Să se afișeze toate cuvintele distincte ce se pot forma prin eliminarea câte unui singur caracter din cuvântul dat.
6. Să se scrie o funcție care convertește un caracter reprezentând o cifră hexazecimală în echivalentul întreg zecimal.
7. Scrieți o funcție: `int vals(int *s, int ds, int *d, int *dd)`; care pune valorile distincte ce apar în vectorul s de dimensiune ds în vectorul d , iar în dd numărul acestor valori; returnează $*dd$.
8. Să se scrie o funcție care verifică dacă un șir de caractere este compus în întregime din cifre. Folosind această funcție, să se scrie un program care citește de la intrare șiruri de caractere (câte unul pe fiecare linie) până la întâlnirea caracterului EOF și afișează inversat (de la dreapta la stânga) fiecare șir citit, exceptând șirurile compuse doar din cifre, care se vor afișa nemodificate.
9. Scrieți funcții pentru citirea unei matrici dreptunghiulare de numere întregi, afișarea unei asemenea matrici și suma a 2 asemenea matrici. Primele două funcții returnează void, a treia returnează un întreg care este 0 în caz de eșec (de exemplu matricile nu au aceleași dimensiuni), altfel succes. Funcțiile vor primi matricile asupra cărora operează ca parametri. De asemenea alocati dinamic memoria pentru matrici.