LUCRAREA 8

Scopul lucrării *este continuarea studiului pointerilor, prin înţelegerea noţiunilor specifice de pointer* NULL*, de aritmetică a pointerilor, a legăturii interne ce există între vectori şi pointeri, împreună cu aspectele legate de alocarea dinamică.*

I. OBSERVAŢII TEORETICE

1.1. Pointeri (*Continuare*)

1.1.1. Pointer NULL

Există o valoare specială, neutră, în situaţia variabilelor de tip pointer. Aceasta este `NULL`, scrisă exact în acest fel, cu majuscule. Definiţia sa este în fişierul header `stdlib.h` şi arată astfel:

```
#define NULL  0
```

sau

```
#define NULL  0L
```

adică valoarea întreagă scurtă fără semn (`0`), sau valoarea unui întreg lung fără semn (`0L`). Sunt două valori, pentru că există variabile care se pot puncta (adresá) în interiorul unui segment de date - `intrasegment`, maxim `64 KB`, sau între segmentele de date (sau de cod) - `extrasegment`, adică peste `64 KB`.

Pentru prima situaţie, sunt de ajuns `65536` valori, adică [`0, 65535`], iar pentru a doua situaţie sunt necesare `4` miliarde de situaţii (exact `4,294,967,296`), în intervalul [`0, 2`^{32} `- 1`].

Este periculoasă folosirea unui pointer `NULL` împreună cu operatorul de indirectare (`*`), pentru că valoarea sa de tip adresă fiind 0, a lucra cu conţinutul de la acea adresă este interzis (există plaje de adresă rezervate de către producătorii hardware/software) pentru scrierea `rutinelor de bază` ale sistemului. Pentru a observa efectele pe care le are o asemenea greşeală renunţaţi la comentariul de pe linia `12` a primului program de studiu din acest laborator (secţiunea ***Desfăşurarea lucrării***, problema 2.1).

1.2. Aritmetica pointerilor

1.2.1. Legătura vector - pointer

Pentru **`orice vector`**, numele acestui reprezintă adresa sa de început, adică adresa de unde începe spaţiul continuu rezervat vectorului. În lucrările 6 şi 7 s-a folosit - inevitabil - **`operatorul de indexare`**, `[i]`. Acest operator aplicat numelui vectorului adresează componenta a `(i+1)`$^{-a}$, în numărătoare de la `0`.

Mecanismul intern prin care compilatorul realizează această operaţie (de indexare) este aritmetica pointerilor. Prin aceasta se înţelege operaţia de `adunare/scădere în mulţimea N` (naturală) a unui număr la un pointer.

Dacă un **`pointer p`** arată la o variabilă de un anumit **`tip de dată t`**, atunci prin adunarea unui **`număr natural n`** (prin generalizare, întreg) la acesta se înţelege saltul în memorie cu un număr de octeţi egal cu:

```
n*sizeof(t)
```

Acest lucru este absolut normal pentru vectori, pentru că spaţiul de memorie este continuu, şi deci pentru a puncta un element cu `index i` din vector trebuie să parcurgem în memorie întregul spaţiu ocupat de toate elementele precedente, adică indecşii `0...i-1`. Adică trebuie să lăsăm deoparte un număr de octeţi egal cu `i*sizeof(tip_vector)`.

Pentru vectorul de întregi:

```
int vector[16];
int *pV = vector;              // pointer variabil
```

următoarele notaţii sunt echivalente:

```
vector[i] ≡ pV[i] ≡ *(pV+i) ≡ *(vector+i)
```

adică, în notaţie cu pointer sunt executate, în următoarea ordine, operaţiile:

- `mai întâi`, se adună un număr întreg la numele pointerului (avans în memorie cu un multiplu al numărului de octeţi cerut de tipul de bază al pointerului);
- `apoi` se foloseşte operatorul indirectare (sau dereferinţă, `*`) pentru obţinerea valorii stocate în cei `sizeof(tip)` octeţi din zona de memorie punctată (adresată), adică obţinerea valorii numerice proprie tipului vectorului, stocată în acea zonă.

Acest mecanism de aritmetică a pointerilor mai poartă numele şi de **`adresare indirectă`**. Indirectă pentru că se ajunge la o zonă de memorie prin intermediul unui pointer, folosind şi operaţia suplimentară de `indexare`.

1.3. Alocare dinamică (aspecte practice)

Limbajul *C* se remarcă, atât pozitiv, cât şi negativ, prin lucrul cu pointeri. Pozitiv pentru că oferă flexibilitate celor ce înţeleg acest mecanism, iar negativ pentru că (şi) din această cauză el este clasificat drept `limbaj de nivel mediu`, existând alte limbaje mai expresive (*SmallTalk*, LISP ş.a.). Pointerii sunt consideraţi frecvent drept o sursă de erori ale unui program care-i foloseşte.

Există două tipuri de alocare de memorie:

- **`statică`** - în momentul *compilării*;
- **`dinamică`** - în momentul *rulării*.

Până acum întregul spaţiu de memorie necesar variabilelor era rezervat automat de către compilator odată cu declararea/definirea variabilelor. NU au existat situaţii în care spaţiul să fie în aşa măsură solicitat, încât compilatorul să ne furnizeze o eroare de alocare (***Not enough memory***). Dar, există în problemele de programare şi situaţii în care este nevoie de spaţiu de memorie relativ mare (de ordinul KB sau zecilor de KB), şi rezumându-ne doar la modalităţile de alocare statică, nu vom putea rezolva acele cerinţe. Două exemple concrete sunt acelea în care se proiectează o bază de date, sau o aplicaţie gen editor/procesor de texte.

În asemenea situaţii intervin funcţiile de alocare pe care le posedă compilatorul. Există două fişiere header dedicate acestor operaţii, anume **`alloc.h`** şi **`stdlib.h`**. Aici sunt cuprinse funcţii proprii stilului de lucru cu pointeri, zone de memorie şi mecanisme de adresare. **`alloc.h`** este posibil să nu fie compatibil pe toate sistemele ce rulează un compilator de *C*. Dacă acest lucru se întâmplă, folosiţi **`stdlib.h`**.

1.3.1. Pointer către `void` (`void *`)

Acesta este un tip generic de dată. `void` are aici semnificaţia de **orice**. Dacă reţineţi, tipul de dată `void` era folosit şi la funcţii, fie ca tip returnat, fie ca argument. Semnificaţia lui acolo este de **nimic**. Adică exact semnificaţia opusă celei din contextul pointerilor. Un asemenea tip de dată este deci puternic, din moment ce poate lua două înfăţişări diametral opuse, relativ la contextul de lucru.

Aşadar, '`pointer la void`' semnifică un pointer care poate puncta către orice tip de dată (fundamental sau compus). Necesitatea acestui tip de dată s-a datorat cerinţelor de generalitate impuse funcţiilor de bibliotecă. Adică o proiectare a lor care să fie valabilă în toate situaţiile de lucru tipice acestora.

1.3.2. Funcţiile de bibliotecă `malloc()` / `calloc()` şi `free()`

Un `pointer la void` se foloseşte şi în cazul funcţiilor de alocare dinamică: `malloc()` (*memory allocation*) şi `calloc()` (*clear memory allocation*).

Tipul de dată returnat de acestea este ' `void *` ', deoarece în cazurile practice se poate aloca memorie pentru virtual ORICARE tip de dată - fundamental (`char`, `int`, `float`, `double`, `long double` ş.a.) sau definit de către utilizator (prin `typedef`).

Pentru că ' `void*` ' semnifică pointer la orice, în cazul concret al folosirii acestor funcţii se vor impune **`conversii explicite de tip`**. Conversiile vor modifică 'pointerul la `void`' în pointer la tipul de dată concret.

Exemplu:

```
typedef int *pInt;
pInt pIntreg;

// alocarea de memorie pentru o variabilă întreagă
pIntreg = (pInt) malloc (sizeof(int));
```

Odată definit tipul de dată 'pointer la întreg', cu ajutorul său s-a declarat un pointer, `pIntreg`. Acesta va puncta în memorie cei doi octeţi (sau 4 octeţi, în funcţie de compilatorul folosit) rezervaţi de `malloc()` unei variabile întregi. Accesul la spaţiul efectiv de memorie şi stabilirea de valori proprii tipului întreg se vor face, din momentul alocării, doar prin intermediul lui `pIntreg`. Avem aici o atribuire: membrul stâng este un pointer la un întreg. Ceea ce întoarce `malloc()` este, conform definiţiei sale, un pointer la `void*`. Dacă am lăsa atribuirea în forma incorectă:

```
pIntreg = malloc (sizeof(int));
```

am primi din partea compilatorului un mesaj de eroare, care se referă la conversia de la `void*` la `int*`. Această conversie nu se poate face automat de către compilator.

C:\BC\MyApp\Laborator\Lab_8\lab8_alocareDinamica1.cpp
***[Warning] In function `int: 14* - invalid conversion**

De aceea, trebuie să apară explicit conversia către `int*`, adică aici paranteza: `(pInt)`. În acest fel, tipurile de date drept şi stâng sunt identice, iar atribuirea este din toate punctele de vedere corectă.

Observaţie:

După ce se foloseşte o zonă de memorie, convenţia este ca aceasta să se elibereze pentru utilizări ulterioare. Funcţia de bibliotecă ce îndeplineşte acest rol este **`free()`**, iar ca argument aceasta primeşte valoarea adresei de unde începe zona alocată prin `malloc()`.

Exemplu:

```
free(pIntreg);
```

În acest fel se eliberează zona de memorie rezervată întregului din exemplul anterior. Acea zonă este de acum 'pusă la comun', făcând parte din mulţimea de adrese `disponibile` unor noi alocări.

Diferenţa dintre cele două funcţii de alocare, `malloc()` şi `calloc()` este acea că cea de-a doua după ce alocă spaţiul de memorie cerut, îl şi curăţă, completându-l cu `0`. Mai există şi o diferenţă în ce priveşte numărul argumentelor. La `calloc()` există două argumente: primul este mărimea tipului de dată, în octeţi, iar cel de-al doilea este numărul dorit de astfel de octeţi.

Prototipul acestei funcţii este:

```
void *calloc(size_t nitems, size_t size);
```

Primul argument semnifică numărul de elemente de un anumit tip, iar cel de-al doilea arată numărul de octeţi al tipului pentru care se doreşte rezervarea de spaţiu. Este şi aici necesară conversia explicită a pointerului `void*` către pointerul la tipul dorit.

1.3.3. Realocarea de memorie: realloc()

Dacă se doreşte folosirea aceluiaşi pointer dealocat anterior, se poate folosi şi funcţie **`realloc()`**. Efectul folosirii acestei funcţii este acela de mărire/micşorarea a spaţiului alocat anterior. Înainte de apelul lui `realloc()` **NU** se foloseşte funcţia `free()`, adică spaţiul nu se de-alocă înainte de a se realoca. Este posibil însă ca adresa de început a noii zone de memorie (mai mică sau mai mare) să nu mai fie aceeaşi cu cea de dinainte.

Prototipul acestei funcţii:

```
void *realloc(void *block, size_t size);
```

Primul argument este pointerul care va marca începutul zonei rezervate, iar al doilea argument numărul de elemente ce se doresc alocate, de tipul la care punctează pointerul de început. Şi aici este necesară conversia explicită în atribuire a pointerului `void*`.

Exemplu:

```
typedef int *pInt;
pInt pIntreg;

// alocarea de memorie pentru o variabilă întreagă
pIntreg = (pInt) malloc (sizeof(int));
…
// re-alocarea de spaţiu de memorie, de această dată pentru două
// variabile întregi
pIntreg = (pInt) realloc (2*sizeof(int));
// mărirea spaţiului realocat la doi întregi
…
```

1.3.4. Alocare de memorie pentru vectori

Alocarea de memorie pentru vectori este un proces î care trebuie să apară toate cele trei entităţi care defineau complet un vector, anume:

```
{nume, tip, nrElemente}
```

Numele vectorului semnifică adresa de început din memorie. Locul său va fi luat acum de un pointer la tipul de bază al vectorului. Tipul de bază al pointerului intră în declaraţia pointerului amintit (de început). Numărul de elemente este argumentul pe care îl preia `malloc()` sau `calloc()`, şi arată

multiplii lui `sizeof(tip)` ce trebuie alocaţi, adică numărul total de octeţi. Acest argument va fi cunoscut doar la rularea programului.

Observaţii:

1. Spaţiul de memorie este **`alocat optim`** în acest caz, fără risipă. În momentul rulării programului, utilizatorul va confirma dimensiunea vectorului, iar alocarea se va face strict pentru numărul de poziţii cerut.
2. Dacă la alocarea statică a vectorilor spaţiul de memorie rezervat era continuu, la alocarea dinamică nu se garantează acest lucru. Astfel, spaţiul de memorie poate fi compus din mai multe zone de memorie, anume acele zone care au fost libere la un moment dat, şi de unde s-au preluat octeţi în numărul cerut la alocare.
3. Ca la lucrul clasic cu vectori, şi aici, pentru atingerea unei componente oarecare `i`, se va putea folosi operatorul de indexare, `[i]`. Exprimarea aceasta fiind echivalentă cu cea de aritmetică a pointerilor, se va putea folosi, la un moment dat, una sau alta dintre aceste variante de notaţie.

Exemplu:

```
...
int i;    // pentru ciclul for() ce urmează
int *pIntreg;
int dim;  // dimensiunea vectorului - cunoscută la rulare

printf("\n Câte elemente să conţină vectorul?"); scanf("%i", &dim);
pIntreg = (int*) malloc(dim*sizeof(int)); // alocarea de spaţiu

for(i=0; i<dim; i++)
printf("\n elementul %i are valoarea: %i", i, pIntreg[i]);
...
```

1.4. Detalii de programare

Pentru înţelegerea programelor utilizate în lucrare studiaţi *Anexa* lucrării 8.

1.5. Comenzile compilatorului

Înainte de compilare codul sursă trebuie salvat (cu combinaţia de taste `CTRL+S`), sau, echivalent, din meniul `File`, cu comanda `Save as...`.

Compilarea se va face cu comanda `CTRL+F9`, iar rularea cu comanda `CTRL+F10`. Compilarea şi rularea se pot executa, succesiv, printr-o singură tastă, anume `F9`. Acţiunile echivalente din meniu: meniul `Execute` comanda `Compile`, respectiv `Compile + Run`.

Observaţii:

1. Orice nouă modificare a codului-sursă trebuie urmată obligatoriu de salvarea acestuia (`CTRL+S`). Abia după aceea pot urma celelalte acţiuni dorite.
2. Dacă nu se realizează aducerea la zi a programului, compilarea şi rularea ce urmează se fac tot pe varianta veche, adică ce de dinaintea ultimelor modificări.

II. DESFĂŞURAREA LUCRĂRII

2.1. Noţiunea de pointer `NULL` este studiată în programul `lab8_1.cpp`. Rulaţi programul în varianta prezentată.

Renunţaţi apoi la comentariul de pe linia `12` şi marcaţi-vă eroarea generată de compilator. De ce apare această eroare? Corectaţi programul.

2.2. `Pointerul la void` este o construcţie foarte versatilă în C. Înţelegerea acestui concept se face cu ajutorul programului `lab8_2.cpp`. Se face aici conversia unui pointer declarat drept `void*` către tipul de dată dorit (către un întreg în exemplul concret din program).

Declaraţi un nou pointer `void*`, iniţializaţi-l şi stabiliţi cu ajutorul lui valoarea variabilei la care punctează.

2.3. Noţiunile legate de alocarea dinamică sunt tratate în programul `lab8_3.cpp`. Sunt folosite pentru alocare funcţiile `malloc()` şi `calloc()`. Se face o alocare (şi, in final se de-alocă spaţiul) de memorie pentru o variabilă întreagă.

Renunţaţi la comentariul de pe linia 14 şi recompilaţi programul. Comparaţi eroarea generată cu cea din prezentarea teoretică, paragraful 1.3.2.

Completaţi programul folosind şi noţiunea studiată anterior de pointer la void*. Declaraţi mai întâi acest pointer generic (deci de tip void*) şi folosiţi-l într-o nouă alocare dinamică, pentru două variabile de tip float. Stabiliţi apoi valorile acestor două variabile alocare la 1.1, respectiv -2.2. De-alocaţi în final spaţiul de memorie şi încercaţi - după de-alocare - să afişaţi conţinutul zonei de memorie dealocate.

2.4. Re-alocarea de spaţiu de memorie este studiată în programul lab8_4.cpp. Se porneşte cu alocarea de spaţiu pentru o variabilă întreagă. Valoarea acestei variabile se stabileşte la 65. Se extinde apoi, prin realocare, acest spaţiu la două variabile de tip întreg. Se stabileşte apoi valoarea celui de-al doilea întreg la 97. Valorile alese nu sunt întâmplătoare. Într-o afişare cu specificator %c (pentru caractere) se observă literele A respectiv a. Valorile alese reprezintă codul ASCII al acestor caractere. Este folosit aici şi operatorul typedef (revedeţi **Lucrarea 4** dacă este necesar).

Înainte de de-alocarea de spaţiu de pe linia 26, cu ajutorul pointerului adunaţi valoarea celui de-al doilea întreg (adică 97) la valoarea primului (adică la 65) şi afişaţi rezultatul folosind aceiaşi specificatori ca în afişările precedente.

III. ÎNTREBĂRI

3.1. Declaraţi următorii pointeri:

- doi pointeri la un întreg
- un pointer către float
- pointer la caracter
- pointer la un şir de caractere.

3.2. Să presupunem că An şi Bn sunt doi întregi. P şi Q sunt pointeri la cei doi întregi. Iniţializăm pe P şi Q cu adresele lui An şi Bn, respectiv. Arătaţi cum înmulţim pe An cu Bn, atribuind rezultatul lui P. Adunaţi apoi cantităţile la care punctează P şi Q, atribuind rezultatul lui Bn.

3.3. Să presupunem că chPtr este un pointer către char, adică: char *chPtr; iar în memorie există următoarea configuraţie a adreselor:

Variabilă	Adresa	Conţinut
chptr	1010	1020
	1020	'L'
	1021	'a'
	1022	' '
	1023	'M'
	1024	'u'
	1025	'l'
	1026	't'
	1027	'i'
	1028	'A'
	1029	'n'
	1030	'i'
	1031	'!'
	1032	'\0'

Determinaţi următoarele valori:

a) **chptr*
b) **chptr+1*
c) **(chptr + 1)*
d) *(*chptr) + 1*
e) *&chptr*
f) **&chptr*

3.4. Scrieţi o instrucţiune de afişare `printf()` în care să afişaţi la ecran adresa unui pointer la caracter, adresa din acea variabilă pointer şi valoarea caracterului la care el punctează.

3.5. Creaţi un program în care, pentru un vector de `8` elemente de tip real simplă precizie, iniţializat în momentul declarării, să afişaţi conţinutul fiecărei componente folosind explicit operaţia de adunare a unui număr întreg la numele unui pointer.

3.6. Alocaţi dinamic spaţiu de memorie pentru `două` variabile de tip caracter. Puteţi folosi oricare dintre funcţiile de alocare. Apoi:

- afişaţi conţinutul imediat după alocare (înainte de a scrie ceva în spaţiul rezervat);
- scrieţi caracterele `'A'` şi `'!'` în cei doi octeţi;
- afişaţi aceste valori;
- dealocaţi în final întreaga zonă rezervată.

3.7. Gândiţi-vă la o situaţie practică în care se impune realocarea ***prin micşorare*** a unei zone de memorie rezervate în prealabil. Scrieţi un program prin care:

a. să rezervaţi la început o zonă de `8` întregi,

b. scrieţi în fiecare componentă o anumită valoare (adică iniţializarea componentelor) folosind notaţia în `aritmetică a pointerilor`,
c. afişaţi adresa din memorie a fiecărei variabile întregi alocate,
d. realocaţi zona de memorie, de această dată doar cu `4` componente,
e. eliberaţi în final întreaga zonă folosită.

3.8. `Alocaţi dinamic` spaţiu de memorie pentru un vector de 24 caractere (caracterul cu index 23 (din cele 24) este '`\0`'). Funcţia de alocare este `calloc()`. Completaţi şirul astfel obţinut cu mesajul "`Vineri este vacanta`". Aduceţi apoi, şirul în varianta: "`Vacanta este Vineri`". Dealocaţi în final, spaţiul de memorie alocat anterior.

3.9. Pentru vectorul următor
float vector[10] = {0, -1, 2, -3, 4, -5, 6, -7, 8};

afişaţi conţinutul componentelor cu indice 4 şi 6 în mod direct (operator de indexare) şi indirect (aritmetica pointerilor).

IV. ANEXA - sursele complete ale programelor

```
// lab8_1.cpp
// Pointer NULL.
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
 char ch = 'W';
 char *pNull = NULL;    // pointer neinitializat. In asteptare...

 printf("\n Semnificatia unui pointer NULL: %p", pNull);

// eroare pe linia următoare (!). Nu pot folosi un pointer NULL (!!)
// *pNull = 'A';

// initializarea pointerului NULL
 pNull = &ch;  // acum pNUll nu mai este NULL. El arata la caracterul ch.
 printf("\n Adresa: %p, valoarea intreaga: %i, si caracterul: %c",
                                        pNull, *pNull, *pNull);
// incheiere
 puts("\n Final..."); scanf("%c", pNull); return 0;
}
```

```
// lab8_2.cpp
// Pointer void.
#include<stdio.h>
int main(void)
{
 void *pVoid;
 int a = 5, *pInt=&a;  // un intreg, si pointerul la el

 (int*) pVoid = &a; // conversia pointerului la void, si stabilirea sa la
                    // aceeaşi adresa cu cea la care arata pint.
 *((int*) pVoid) = *pInt + 2;
 printf("\n Valoarea lui a: %i, sau %d", *((int*)pVoid), *pInt);

 *(int*) pVoid = 9;

 printf("\n Valoarea lui a: %i, sau %d", *((int*)pVoid), *pInt);

// incheiere
 puts("\n Final..."); scanf("%c", pNull); return 0;
}


// lab8_3.cpp
// Alocare dinamică - malloc() şi calloc().
#include<stdio.h>
#include<stdlib.h>

typedef int *pInt;     // tipul de data 'pointer la intreg'

int main(void){
 pInt pIntreg;         // o variabila de tipul definit mai sus

// Alocarea de memorie pentru o variabila întreaga - malloc()
 pIntreg = (pInt) malloc (sizeof(int));
// pIntreg = malloc (sizeof(int));
 printf("\n malloc(): Valoarea din memorie, inainte de \
          initializare: %i, %x", *pIntreg, *pIntreg);

//  Stabilirea de valori pentru variabila intreaga alocata,
// si afisarea acesteia
//  In memorie s-a scris pe cei doi octeti rezervati, valoarea
// binara 20, adica 14 hexa.
 *pIntreg = 20;
 printf("\n Adresa in memorie: %u, %p", pIntreg, pIntreg);
 printf("\n Valoarea din memorie: %i, %x(hexa)", *pIntreg, *pIntreg);

// Dealocarea (eliberarea spatiului) alocat anterior
 free(pIntreg);
 printf("\n Adresa in memorie, dupa dealocare: %u, %p", pIntreg, pIntreg);

// Alocarea de memorie pentru o variabila întreaga - calloc()
 pIntreg = (pInt) calloc (sizeof(int), 1);

 printf("\n\n calloc(): Valoarea din memorie, inainte de initializare: %i,\
```

```
          %x", *pIntreg, *pIntreg);
//  Stabilirea de valori pentru variabila intreaga alocata, si
// afisarea acesteia
//  In memorie s-a scris pe cei doi octeti rezervati, valoarea
// binara 40, adica 28 hexa.
 *pIntreg = 40;
 printf("\n Adresa in memorie: %u, %p", pIntreg, pIntreg);
 printf("\n Valoarea din memorie: %i, %x(hexa)", *pIntreg, *pIntreg);

// de-alocarea (eliberarea spatiului) alocat
 free(pIntreg);
 printf("\n Adresa in memorie, dupa dealocare: %u, %p", pIntreg, pIntreg);

// incheiere
 puts("\n Final..."); scanf("%c", pNull); return 0;
}


// lab8_4.cpp
// Realocarea de spaţiu de memorie.
#include<stdio.h>
#include<stdlib.h>

typedef int *pInt;

int main(void)
{
 pInt pIntreg = (pInt) calloc(sizeof(int), 1) ;   // un intreg

 printf("\n calloc(): Adresa de memorie: %p", pIntreg);
 *pIntreg = 65;  // codul ASCII al literei 'A'
 printf("\n %c, %d, %x", *pIntreg, *pIntreg, *pIntreg);

// re-alocarea spatiului de memorie, prin extinderea sa la doi intregi
 pIntreg = (pInt) realloc(pIntreg, 2);

 printf("\n\n realloc(): Adresa de memorie: %p", pIntreg);
 *pIntreg = 65;
 *(pIntreg+1) = 97;           // codul ASCII al lui 'a'
 printf("\n Primul intreg: %c, %d, %x", *pIntreg, pIntreg[0], *pIntreg);
 printf("\n Al doilea intreg: %c, %d, %x",
                              *(pIntreg+1),              pIntreg[1],
                              *(pIntreg+1));
// de-alocarea (eliberarea zonei de memorie folosite)
 free(pIntreg);  // eliberarea spatiului rezervat - acum 2 intregi

// incheiere
 puts("\n Final..."); scanf("%c", pNull); return 0;
}
```

```
// Sursă suplimentară - lab8_5.cpp
// Alocare vector.
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
 int i;          // pentru ciclul for() ce urmeaza
 int *pIntreg;
 int dim;        // dimensiunea vectorului - cunoscuta la rulare

 printf("\n Câte elemente sa contina vectorul?"); scanf("%i", &dim);

// alocarea efectiva
 pIntreg = (int*) malloc(dim*sizeof(int)); // alocarea de spatiu

// parcurgerea vectorului
 for(i=0; i<dim; i++)
      printf("\n elementul %i are valoarea: %i", i, pIntreg[i]);

// de-alocarea spatiului
 free(pIntreg);

// incheiere
 puts("\n Final..."); scanf("%c", pNull); return 0;
}
```