Variabile de tip adresă, pointeri

Considerăm următoarea declarare:

```
int x;
```

Este o variabilă de tip `int`, adică valoarea sa este un număr întreg pe 32 de biți (așa se memorează în C/C++ o dată de tip `int`). Ce trebuie să aibă în minte programatorul când scrie o astfel de linie de cod ? Răspunsul este următorul: atunci când programul de lansează în executare, în memoria RAM se va rezerva o zonă unde se va stoca o valoare de tip `int`. Zona rezervată pentru o variabilă de tip `int` este de patru octeți (deci 32 de biți). Acești patru octeți vor fi vecini în memoria RAM. Acest lucru este valabil pentru orice variabilă, chiar pentru variabilele compuse. De exemplu, o declarare de forma `short v[10]`, va avea ca efect rezervarea unei zone de `10X2 = 20` octeți conecutivi.
Pe parcursul executării unui program, memoria RAM pusă la dispoziția acestuia ne-o imaginăm ca pe un șir de octeți. Fiecare octet are o poziție în acest șir, iar această poziție o vom numi adresă. Adresa de memorie unde se află o variabilă se consideră adresa *primului* octet al zonei continue alocate acelei variabile.

Există posibilitatea declarării de **variabile a căror valoare să fie chiar o adresă de memorie**. Ele se mai numesc și **pointeri**. Aceste variabile, la declarare, se asociază de regulă cu un tip de date, idicând ce fel de dată dorim să accesăm la adresa memorată în pointer. Modul de declarare pentru un pointer este următorul:

```
tip *nume;
```

Exemple

```
int *p;
char *q
long long *r;
```

Fiecare dintre cele trei variabile declarate mai sus poate stoca o adresă de memorie, însă anterior am spus că o adresă de memorie este de fapt poziția din memorie a unui singur octet. Din acest punct de vedere, fiecare dintre cele trei sunt la fel, stochează adresa unui singur octet. Care este însă diferența? Prin intermediul pointerului vom putea accesa valoarea efectivă de la adresa memorată în pointer. Și aici intervine tipul de date indicat. Dacă pointerul este de tip int, accesarea valorii de la acea adresă se va face verificând patru octeți începând cu cel memorat în pointer, dacă pointerul este de tip long long, se verifica opt octeți începând cu acela memorat în pointer etc.

Sintaxa accesării valorii de la adresa memorată în pointer este următoarea:

`*nume`	În acest exemplu, `nume` reprezintă numele variabilei de tip pointer.

Trebuie remarcată semnificația oarecum diferită a operatorului `*`, în funcție de context. Întâi a precedat numele variabilei la declarare, stabilind tipul pointer, apoi permite ca printr-un pointer să avem acces la zona de memorie cu adresa memorată în acel pointer.

Pentru a putea prezenta următoarele exemple, avem nevoie de un operator nou, operatorul `&`, numit operator adresă. Putem spunem că este invers al operatorului `*`, dacă `*` permite să accesăm o valoare când precede un pointer, operatorul `&` permite să obținem o adresă când precede o variabilă obișnuită.
Exemplu:

```
int x;
int *p;
x = 2;
```

```
p = &x;
cout << *p;
```

Secvența de mai sus reprezintă un cod corect în C/C++, și o interpretăm astfel: pe linia unu am declarat o variabilă de tip `int`, pe linia a doua am declarat o variabilă ce poate memora adresa unei zone de memorie unde se stochează un `int`. Instrucțiunea de pe linia a treia este o atribuire cunoscută, în zona de memorie asociată cu numele `x` se va memora valoarea 2. Linia a patra aduce un element de noutate, anume că operatorul de atribuire (=)este permis și pentru pointeri. Efectul este, evident, copierea adresei obținute în urma evaluării expresiei din dreapta sa, în variabila pointer din stânga operatorului sa. Deci, pointerul `p` va memora adresa variabilei `x`. Ultima linie are ca efect afișarea pe ecran a valorii 2. Observăm că avem o singură zonă de memorie unde putem stoca un `int`, dar acea zonă o putem accesa în două moduri, cel obișnuit, prin numele `x`, dar și prin `*p` dacă anterior am stocat în `p` adresa lui `x`. Ce se afișează ca urmare a rulării secvenței următoare (am adăugat două linii de cod)?

```
int x;
int *p;
x = 2;
p = &x;
cout << *p;
*p = 3;
cout<<x<<*p
;
```

Răspunsul este: `233`. Așadar, odată ce se execută `*p = 3`, valoarea `3` este stocată de fapt în `x`, căci `p` memorează adresa lui `x`.

Nu putem face atribuiri între pointeri de tipuri diferite. Exemple:

<table>
<tr><td colspan="2"><pre>int x;
int *p;
long long y;
long long *r;</pre></td></tr>
<tr><td>p = &x;</td><td>Permis</td></tr>
<tr><td>r = &y;</td><td>Permis</td></tr>
<tr><td>p = &y;</td><td>Nepermis</td></tr>
<tr><td>r = &x;</td><td>Nepermis</td></tr>
<tr><td>*r = *p;</td><td>Aici este permisă instrucțiunea, nefiind vorba de atribuire de pointeri ci se atribuie un int la o variabilă de tip long long.</td></tr>
</table>

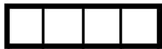
Tablouri

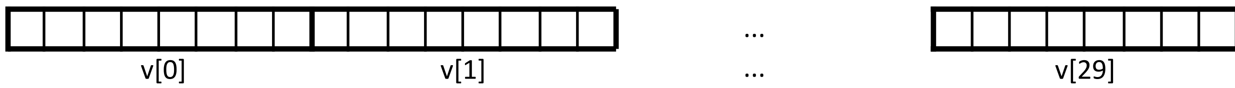
Fie declarația:

```
double
v[30];
```

Am folosit o astfel de declarație și ne este cunoscută semnificația. Este vorba despre 30 de variabile de tip real, numite `v[0]`, `v[1]`, ..., `v[29]`. Putem spune totodată că avem o zonă de `30X8 = 240` de octeți consecutivi (o variabilă de tip `double` ocupă opt octeți). Ceea ce vom sublinia acum este următorul lucru: `v`, numele tabloului, este de fapt o variabilă pointer. Este un pointer la `double`. `v` memorează, ca orice pointer adresa primului octet al zonei de date alocate tabloului. Încă un element de noutate: orice pointer ocupă patru octeți de memorie (deci atât variabila `v` de mai sus cât și celelalte: `p`, `q`, `r` ... folosite anterior ocupă câte patru octeți).
Iată în continuare o reprezentare grafică a ceea ce se află în memorie în urma declarației de mai sus:

v:

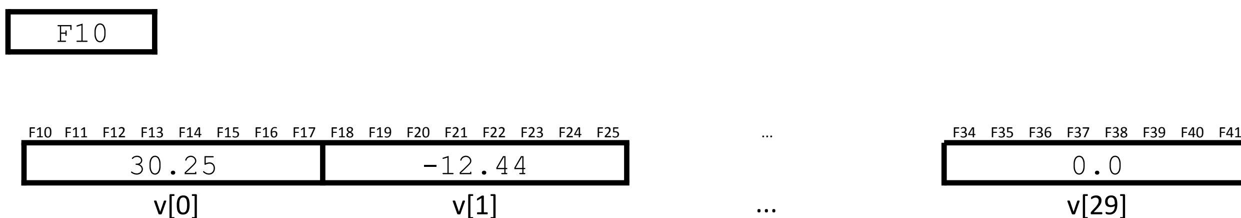




În cei 4 octeți ai lui `v` se memorează un cod ce reprezintă adresa primului octet din zona de date alocată valorilor din tablou, adică a primului octet al lui `v[0]`.

Vom vedea, mai departe, diferite operații pe care le putem realiza cu pointerii. Nu sunt definite operații de citire sau de afișere a pointerilor. Nici nu ne interesează efectiv cum arată valoarea unei adrese. Vom face însă o convenție de notare a unei adrese, în scopul de a explica mai ușor anumite lucruri. Convenția pe care o vom stabili nu o folosim în cod, ci doar în notațiile din desenele utilizare. Vom nota o adresă ca pe un număr în baza 16, care începe cu o cifră de la `A` la `F`. Am ales această convenție de notare și pentru că diferite utilitare afișează de regulă adresele de memorie folosind scriere hexazecimală a anumitor numere. Conform cu cele spuse, ne putem imagina modelul de memorie al declarației de mai sus, astfel:

v:



Tabloul de mai sus memorează în componenta de pe poziția `0` valoarea `30.25`, în componenta de pe poziția `1` valoarea `-12.44` iar în ultima componentă memorează valoarea `0`.

Este deci important să reținem că `v` este un pointer la `double`.
Fie, declarațiile:

```
double v[30];
double *p;
```

Conform celor spuse anterior, `p` și `v` sunt pointeri de același tip. Este permisă atribuirea `p = v;`

Odată cu ea, avem doi pointeri prin intermediul cărora putem accesa elementele vectorului (atenție, însă, că avem un singur vector).

Fie, în continuarea declarațiilor anterioare, următorul cod:

```
int n, i;
cin>>n;
for (i=1;i<=n;i++)
    cin>>v[i];
p = v;
for (i=1;i<=n;i++)
    cout << v[i]<<" ";
cout<<"\n";
```

```
for (i=1;i<=n;i++)
    cout << p[i]<<" ";
cout<<"\n";
```

Dacă introducem de la tastatură 3 apoi 12, 20, 17, efectul este că se va afișa de două ori șirul introdus.

Cu toate că ambele moduri de declarare din exemplul de mai sus anunță pointeri la double, există și diferențe între ele care trebuie bine înțelese.

Declararea lui v anunță deodată două lucruri: pointerul (cei patru octeți) și zona de date (cei 240 de octeți). Declararea lui p anunță doar alocarea celor patru octeți pentru pointer. Adresa zonei alocate pentru date se memorează în v și din acest motiv nu ne este permisă modificarea valorii pointerului v până la finalul programului. Așadar, dacă o atribuire de forma p = v am văzut că este permisă, o atribuire de forma v = p va produce eroare de compilare.

Deci, o declarare ca v provoacă si alocarea unei zone de memorie pentru date, pe langa cei patru octeți pentru pointer, iar o daclarare ca a lui p provoacă doar alocarea de patru octeți de memorie, unde se poate pune ulterior o adresă. În limbajul C/C++ putem folosi constanta NULL pentru a indica un pointer vid (nicio adresă).

În urma declarațiilor de mai sus este permisă o atribuire p = NULL, dar, așa cum am spus, nu putem scrie v = NULL, pentru că nu este permisă schimbarea adresei din v. Pointerii declarați ca și v se numesc pointeri constanți.

Iată, în continuare, alte operații cu pointeri:

- adunarea unui pointer cu un număr

Fie secvența de cod:

```
int v[30];
int *p;
v[0] = 20;
v[1] = 30;
v[2] = 100;
p = v;
cout<<*p<<"\n"
;
```

Se va afișa 20, adică valoarea lui v[0], pentru că, p fiind un pointer la int, *p reprezintă valoarea unei zone de tip int care începe la octetul cu adresa memorată în p. Tot la fel de bine am fi putut scrie cout<<*p, cu același efect;
Modificăm secvența anterioară adăugând: p = v+2; cout<<*(v+1)<<" "<<*p;

```
int v[30];
int *p;
v[0] = 20;
v[1] = 30;
v[2] = 100;
p = v;
cout<<*p<<" ";
p = v+2;
cout<<*p<<" "<<<<*(v+1);
```

Întreg codul de mai sus este corect, și se va afișa 20 100 30. În două locuri avem operatorul + care are în stânga un pointer și în dreapta un număr. În ambele cazuri observăm că rezultatul este un pointer. Așadar, adunând un număr la un pointer, obținem tot un pointer. Adresa obținută este peste atâtea componente de

tipul pointerului după pointerul din stânga lui +, nu peste atâția octeți. Deci, p fiind pointer la int, p+1 reprezintă adresa acelui octet de după 4 octeți de la cel a cărui adresă este memorată în p, iar dacă p ar fi reprezentat pointer la char, p+1 ar fi fost chiar după un octet (căci datele de tip char ocupă un octet).

Instrucțiunea p=v+2 se execută astfel: se realizează operația din dreapta lui =; valoarea obținută (o adresă) se memorează în variabila din stânga lui =.

Instrucțiunea cout<<*(v+1) se execută astfel: se aplică operatorul + între pointerul v și numărul 1, se obține o adresă (a octetului aflat peste 4 octeți de la cel a cărui adresă este memorată în v), iar valoarea (de tip int) de la această adresă (formată din 4 octeți începând cu cel a cărui adresă a fost obținută) este afișată.

- aplicarea operatoruilor ++ și −− la o variabilă de tip pointer.

p++ este echivalent cu p = p+1; (același efect îl obținem folosind ++p)

p−− (ca și −−p) are ca efect copierea în p a adresei (octetului de început al) componentei anterioare celei cu adresa pusă curent în p.

- aplicarea operatorului − între doi pointeri

Rezultatul este un int, și anume numărul de componente dintre cei doi pointeri. Astfel, dacă în contextul declarațiilor de mai sus avem p = v+3, atunci p−v dă ca rezultat valoarea 3.

- compararea pointerilor

Se pot pune între doi pointeri operatori relaționali. De regulă folosim == și != pentru a testa dacă doi pointeri memorează aceeași valoare.

Exemplu:

```
int x, y, *p, *r;
p = &x;
r = &y;
x = 2;
y = 2;
cout<<(*p == *r) // se afiseaza 1, adica adevarat, pentru ca nu s-au
                 // comparat pointerii ci valori de tip int
cout<<(p==r); // se afișează 0, adică fals, întrucât cei doi pointeri
              // memorează adrese diferite, chiar dacă la ambele
              // adrese este aceeași valoare (aici operatorul == a
fost
              // folosit între doi pointeri)
p = r;
cout<<(*p == *r)<<(p==r); // acum, evident se afiseaza 11 ambii
                          // pointeri stocand aceeasi adresa
```

În continuare vom folosi operațiile cu pointeri în exemple ce arată diverse moduri de a afișa elementele unui tablou unidimensional. Considerăm că avem:

```
int v[100], *p, n, i;
cin>>n;
for (i=1;i<=n;i++)
    cin>>v[i];
```

Fiecare din secvențele următoate tipărește elementele tabloului care se află pe pozițiile 1, 2, ... n.

Cod	Explicație
`for (i=1;i<=n;i++)` `    cout<<v[i];`	Afișarea folosind secvența clasică de cod
`for (i=1;i<=n;i++)` `    cout<<*(v+i);`	Ne-am folosit de adunarea de număr la pointer pentru a obține adresa fiecărei locații din tablou. Reținem deci echivalența între `v[i]` și `*(v+i)`.
`for (p=v+1;p!=v+n+1;p++)` `    cout<<*p;`	Aici am folosit un pointer auxiliar care ia pe rând adresa fiecărei componente a tabloului și accesăm componenta folosind operatorul `*`.
`for (p=v+1;p!=v+n+1;p++)` `    cout<<p[0];`	Același lucru ca mai sus, dar am folosit echivalența între `p[0]` și `*p`.

Alocarea dinamică a memoriei

În exemplele anterioare am arătat multe dintre instrumentele pe care le avem la dispoziție înlucrul cu pointeri. Totuși, din felul în care i-am utilizat nu se deduce că ar fi neapărat nevoie de ei, ci doar că prin intermediul lor putem face lucrurile și altfel. Dar pointerii reprezintă un puternic instrument pentru a face lucrurile mai bine din punct de vedere al memoriei ocupate de program dar și din punct de vedere al timpului de executare, în multe cazuri.

De regulă variabilele pe care le declarăm sunt alocate în memorie pe toată durata executării blocului în care sunt declarate, chiar dacă uneori nu mai sunt începând cu un anume punct din program. De exemplu, dacă pe o zonă de cod avem nevoie de un tablou cu date de tip int, iar de la un moment dat nu mai este necesar acesta ci unul cu date de tip double, în partea a doua noi vom avea memoria ocupată cu ambele tablouri, chiar dacă primul nu mai este necesar. Ați putea spune că rezolvarea era folosirea de la început a unui tablou real, dar asta ar fi dus la un cod mai lent în prima parte, acolo unde se lucra cu întregi. Evident se găsesc și alte motivări.

Spunem că până acum datele declarate de noi au fost statice (în sensul în care nu mai putem să ne eliberăm de ele cân dorim noi). Limbajul C/C++ pune la dispoziție operatori care permit să alocăm memorie în timpul executării, atunci când decidem că avem nevoie, dar și să eliberăm memorie, dacă aceasta decidem că nu mai este necesară. Este esențial să reținem că putem elibera la execuție doar memori alocată de noi, nu pe cea alocată static. Lucrul în acest mod se cheamă alocare dinamică a memoriei. Vom prezenta modul în care beneficiem de asta în limbajul C++, pentru limbajul C principiile de urmat sunt aceleași, însă se utilizează alte elemente de sintaxă (ne vom ocupa de asta altă dată).

În centrul discuției sunt doi operatori: `new` și `delete`.

Pentru a aloca dinamic o zonă în vederea stocării unei date de un anumit *tip*, folosim sintaxa:

`new` *`tip`*

Efectul este următorul: se caută în memorie o zonă continuă, liberă, suficientă stocării unei date de tipul indicat, iar dacă se găsește re returnează *adresa* ei. Așadar, o astfel de scriere este esențial să fie parte a unei instrucțiuni de atribuire către un pointer de tipul indicat, pentru ca adresa zonei alocate să fie copiată într-un pointer. Astfel, putem ulterior elibera această zonă folosind operatorul delete, prin sintaxa:

`delete` *`pointer`*

Deducem că pentru a lucra cu zone alocate dinamic este necesar să lucrăm cu pointeri întrucât operatorii care alocă și eliberează astfel de zone lucrează cu ele prin intermediul pointerilor.

Programatorul trebuie să aibă evidența zonelor pe care le-a alocat dinamic întrucât tot el trebuie să se ocupe de eliberarea lor.

Exemplu:

```
int *p;
p = new int;
*p = 10;
cout<<*p;
delete p;
```

Linia 1 are ca efect crearea unei variabile (statice!) de tip pointer. Pe moment p nu memorează o adresă sub controlul nostru (probabil memorează `NULL`, mai ales dacă este declarată global). Linia a doua are ca efect crearea unei variabile dinamice, iar adresa sa de memorie este stocată în pointerul `p`. Acum se poate rula atribuierea de pe linia a treia, întrucât în `p` este adresa unei zone alocate. Instrucțiunea de pe ultima linie eliberează zona dinamică alocată cu `new`. `p` poate să memoreze în continuare adresa acelei zone, dar aceasta fiind declarată liberă poate să fie rezervată în continuare la altă cerere de alocare de memorie.
Am folosit anterior "adresă de memorie". Reamintesc, este vorba de primul octet al unei zone continue de memorie, însă vom folosi și mai departe această referire generală.

Iată câteva exemple ce ajută la o mai bună înțelegere a lucrului cu pointerii.

Secvență de cod	Efect

`int x, *p;`
`p = &x;`
`delete p;`
	Chiar dacă nu va apărea eroare de compilare, este vorba de o acțiune ilegală care se încearcă la executare deoarece `p` memorează adresa unei zone statice, care nu poate fi eliberată cu `delete`.

`int *p, x;`
`p = new int;`
`p = &x;`
	Nici în acest caz nu avem eroare de compilare dar ultima linie de cod face ca adresa zonei alocate dinamic (cu `new`) să nu se mai găsească niciunde și astfel să nu o mai putem elibera.

`int *p, *r;`
`p = new int;`
`r = new int;`
`p = r;`
`delete p;`
`delete r;`
	Executarea în acest context a instrucțiunii `p = r` produce deja o eroare logică pentru că adresa primei zone alocate dinamic nu o mai avem pusă niciunde.
	Instrucțiuea `delete p` are astfel ca efect eliberarea zonei alocate la al doilea apel `new`. În plus, instrucțiuena delete r va avea ca efect încercarea de eliberare tot a acestei zone, care este deja liberă, lucru care trebuie evitat.

Alocarea dinamică a tablourilor

Evident că alocarea dinamică doar pentru câte o variabilă simplă, ca în exemplele de mai sus, ne-ar putea face să credem că este necesar să declarăm (sau să stocăm) câte un pointer pentru fiecare zonă de dată simplă pe care dorim să o alocăm. Din fericire nu suntem limitați la asta în lucrul cu pointerii.

Să analizăm codul următor:

```
int *p;
p = new int[30];
```

Elementul de noutate care apare este reprezentat de apariția unei dimensiuni (30) între parantezele drepte. Efectul este următorul: Se caută o zonă liberă continuă de 30 * sizeof(int) = 240 de octeți și adresa sa este returnată. Evident mai departe este stocată în pointerul p. Practic, este ca și cum am fi declarat un tablou unidimensional cu elemente de tip int, dar pe care îl putem și șterge (eliberăm memoria) dacă dorim. Este corect în continuare următorul cod:

```
for (int i=0;i<10;i++)
```

```
    cin>>p[i];
```

În secvența de mai sus observăm că unele zone dintre cele 30 alocate nu sunt folosite de noi. Valorile inițiale ale acestor zone nu trebuie să ne bazăm că sunt 0. Ca și variabilele declarate în interiorul unor funcții, acestea nu au o valoare inițială pe care să ne bazăm (de regulă preiau valoarea obținută cu secvența de biți din zona unde se alocă).

Eliberarea memoriei pentru o zonă alocată ca mai sus se face cu următoarea sintaxă:

```
delete []pointer;
```

Așasar, întregul cod de mai sus s-ar scrie corect:

```
int *p;
p = new int[30];
for (int i=0;i<10;i++)
    cin>>p[i];
delete []p;
```

Putem aloca un număr de elemente în funcție de o variabilă citită, deci expresia dintre parantezele ce succed pe new nu trebuie să fie neapărat constantă:

```
#include <iostream>
using namespace std;
int *p, i, n;
int main () {
    cin>>n;
    p = new int[n];
    for (i=0;i<n;i++)
        cin>>p[i];
    for (i=0;i<n;i++)
        cout<<  *(p+i) <<" ";
    delete []p;
    return 0;
}
```

Tablou de pointeri

Anterior am vazut că putem declara un pointer și de la adresa pusă în el să accesăm mai multe componente. În continuare este vorba despre altceva: un tablou de pointeri, adică un vector cu elemente de tip pointer.

Pentru exemplificare, fie următorul enunț: Să se memoreze mai multe șiruri de numere. Se citește mai întâi numărul de șiruri apoi datele fiecărui șir. În cadrul unui șir se citește mai întâi numărul său de elemente și apoi elementele.

Exemplu:

Date de intrare
3 3 1 9 8 4 2 8 7 1 2 1 3

```
#include <iostream>
using namespace std;
int *p[30], i, j, n, L[30];
int main () {
    cin>>n;
    for (i=0;i<n;i++) {
        cin>>L[i];
        p[i] = new int[ L[i] ];
        for (j=0;j<L[i];j++)
            cin>>p[i][j];
    }
    for (i=0;i<n;i++) {
        for (j=0;j<L[i];j++)
            cout<<p[i][j]<<" ";
        cout<<"\n";
    }
    for (i=0;i<n;i++)
        delete []p[i];
    return 0;
}
```

Observăm că în exemplu este vorba despre o "matrice" în care nu toate liniile au același număr de elemente, iar noi am alocat pentru fiecare linie exact câtă memorie trebuie. Deci avem mai sus un tablou de pointeri, iar fiecare pointer este un alt tablou de inturi. Așa cum alocarea de memorie s-a făcut pentru fiecare linie (pointer) separat, și eliberarea se face similar.

Observăm că în exemplu ne-am bazat pe un număr maxim cunoscut de linii (30). Avem posibilitatea să alocăm memorie mai general, fără să ținem cont de acest lucru.

```
#include <iostream>
using namespace std;
int **p;
int *L;
int n, i, j;
int main () {
    cin>>n;
    p = new int *[n]; // cauta o zona de n locatii de tip pointer
                      // la int p este un vector si fiecare componenta
                      // a sa este un pointer la o zona de tip int
    L = new int[n];
    for (i=0;i<n;i++) {
        cin>>L[i];
        p[i] = new int[ L[i] ]; // in fiecare pointer din p pun adresa
                                // unei zone de mai multe inturi (cat
                                // are acel sir)

        for (j=0;j<L[i];j++)
            cin>>p[i][j];
    }
    for (i=0;i<n;i++) {
        for (j=0;j<L[i];j++)
            cout<<p[i][j]<<" ";
```

```
        cout<<"\n";

    }
    for (i=0;i<n;i++)
        delete []p[i];
    delete []p;
    delete []L;
    return 0;
}
```

Se observă că nu intevine nicio valoare constantă, inclusiv numărul de șiruri este preluat de la intrare și se alocă o zonă cu atâția pointeri cât este valoarea dată.

Atenție la ordinea instrucțiunilor din momentul alocării și apoi din momentul eliberării memoriei.

Să extindem puțin exemplul de mai sus pentru a pune mai bine în evidență utilitatea pointerilor. Formulăm următoarea cerință: Să se afișeze șirurile date în ordine lexicografică (amintim, compararea lexicografică este o extensie la șiruri de numere a comparării alfabetice de la cuvinte, adică decizia comparării este la prima poziție din stânga unde diferă caracterele).

Codul funcției de comparare îl prezentăm în continuare:

```
int comparare(int *v, int dv, int *w, int dw)
{
    for (int i=0; i<dv && i<dw; i++)
        if (v[i] < w[i])
            return 1;
        else
            if (v[i] > w[i])
                return 2;
            else
                return 0;
    if (i == dv && i == dw)
        return 0;
    if (i == dv)
        return 1;
    return 2;
}
```

Se compară lexicografic șirul v de lugime dv cu șirul w de lungime dw. Funcția returnează 1 dacă este mai mic lexicografic primul șir, 2 dacă este mai mic al doilea șir respectiv 0 dacă șirurile sunt identice.

Observăm că putem trimite către funcții pointeri, scrierea parametrilor sub forma int*v fiind echivalentă cu int v[], lucru cunoscut de la capitolul despre funcții și parametii lor de tip tablou (când s-a subliniat faptul că atunci când avem parametri tablouri se transmite către funcție doar adresa, fășă să se facă o copie a întregului tablou, ca la parametrii transmiși obișnuit prin valoare).

```
#include <iostream>
using namespace std;
int **p;
int *L;
int n, i, j;
int comparare(int *v, int dv, int *w, int dw) {
    for (int i=0; i<dv && i<dw; i++)
        if (v[i] < w[i])
```

```
            return 1;
        else
            if (v[i] > w[i])
                return 2;
            else
                return 0;
    if (i == dv && i == dw)
        return 0;
    if (i == dv)
        return 1;
    return 2;
}
int main () {
    cin>>n;
    p = new int *[n];
    L = new int[n];
    for (i=0;i<n;i++) {
        cin>>L[i];
        p[i] = new int[ L[i] ];
        for (j=0;j<L[i];j++)
            cin>>p[i][j];
    }
    for (i=0;i<n-1;i++)
        for (j=i+1;j<n;j++) {
            int r = comparare(p[i], L[i], p[j], 
L[j]);
            if (r == 2) {

                ...

            }
        }
    for (i=0;i<n;i++) {
        for (j=0;j<L[i];j++)
            cout<<p[i][j]<<" ";
        cout<<"\n";
    }
    for (i=0;i<n;i++)
        delete []p[i];
    delete []p;
    delete []L;
    return 0;
}
```

Compararea lexicăgrafică face număr de pași de ordinul lungimii șirurilor. Acolo unde sunt punctele de suspensie trebuie să punem codul care realizează interschimbarea. Am putea face asta intershimbând element cu element valorile din cele două zone de date, fiind deci necesar timp de ordinul lungimii șirurilor, sau putem face interschimbarea în timp constant, folosind codul următor:

```
int *paux = p[i];
p[i] = p[j];
p[j] = paux;
int laux = L[i];
L[i] = L[j];
```

```
L[j] = laux;
```

Observăm că este vorba doar de o interschimbare de pointeri și una de inturi. Practic, numerele rămân în zonele de memorie în care au fost citite, interschimbarile făcându-se doar între elementele vectorului de pointeri (evident că odată cu ele sunt necesare actualizări în vectorul de lungimi).

Putem folosi și pointeri la structuri dar cu aceștia vom lucra foarte mult la capitolul dedicat listelor înlănțuite alocate dinamic.

Tablouri cu indici negativi

Știm că în C/C++ tablourile au indici începând cu `0`. În continuare, vom folosi un tablou cu `200` de elemente de tip `int`, cu indici cuprinși între `–200` și `199`. Ne folosim de operația de adunare a unui pointer cu un număr.

```
int v[200];
int *p;
p = v+100;
```

Astfel, scriind `p[0]`, este totuna cu `v[100]`, iar scriind `p[-1]` este totuna cu `v[99]`.