κεφάλαιο

12

Πίνακες ΙΙ – Βέλη

Ο στόχος μας σε αυτό το κεφάλαιο:

Να μάθεις να χρησιμοποιείς πολυδιάστατους (συνήθως δισδιάστατους) πίνακες και βέλη. Με τα βέλη μόλις αρχίζουμε...

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να χρησιμοποιείς πολυδιάστατους πίνακες στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *πολυδιάστατος πίνακας*
- *αποθήκευση στοιχείων πίνακα*
- *παράμετρος-πίνακας*
- *βέλος* (*pointer*)
- *αριθμητική βελών*
- *τυποθεώρηση* `const`
- *παράμετροι της* **main**

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Μέχρι τώρα, στο Μέρος Α, μάθαμε να χρησιμοποιούμε −και χρησιμοποιήσαμε− μονοδιάστατους πίνακες. Φυσικά, σε πολλές περιπτώσεις χρειαζόμαστε πολυδιάστατους πίνακες· συνήθως δισδιάστατους.

Δυνατότητα για χρήση τέτοιων πινάκων μας δίνουν όλες οι γλώσσες προγραμματισμού. Εδώ όμως η C++ (και η C) έχουν μια ιδιαιτερότητα: Για να μπορέσεις όμως να τους αξιοποιήσεις πλήρως θα πρέπει να ξέρεις τα «μυστικά» της υλοποίησής τους.

Το βασικό εργαλείο για τον σκοπό αυτόν είναι το **βέλος** (pointer). Θα αρχίσουμε λοιπόν να το μαθαίνουμε καλύτερα και −καλώς ή κακώς− θα το βρούμε μπροστά μας πολλές φορές στη συνέχεια,

12.1 Πίνακες και Βέλη

Στην §10.13 λέγαμε ότι αν ορίσεις:

```
char a[] = { 'π','ά','ν',' ','μ','έ','τ','ρ','ο','ν',' ',
             'ε','κ','α','τ','ό',' ','π','ό','ν','τ','ο','ι','\0' };
```

και δώσεις:

```
cout << a << endl;
```

θα δεις στην οθόνη σου:

```
πάν μέτρον εκατό πόντοι
```

Και τι θα γίνει αν, ας πούμε, δηλώσεις:

```
double x[] = { 0.1, 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9 };
```

και δώσεις:

```
cout << x << endl;
```

Έκπληξη! Να το αποτέλεσμα:

```
0x22ff20
```

(ή κάτι παρόμοιο.)

Τι είναι αυτό; Αλλά, κάτι μας θυμίζει... Είχαμε δει κάτι τέτοιο στην §2.7.1. Λέγαμε ότι πρόκειται για μια ακέραιη τιμή γραμμένη στο δεκαεξαδικό σύστημα, μια *τιμή-βέλος*, που δείχνει μια *διεύθυνση* (θέση) της μνήμης. Λες να είναι το ίδιο πράγμα; Ας ζητήσουμε τη διεύθυνση του πρώτου στοιχείου (x[0]) του πίνακα· θα χρησιμοποιήσουμε τον τελεστή "`&`":

```
cout << &(x[0]) << endl;
```

Αποτέλεσμα:

```
0x22ff20
```

Η ίδια τιμή!

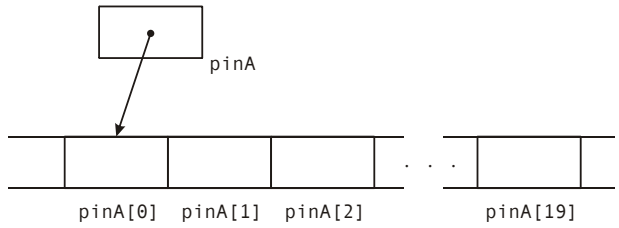
Κάτι παρόμοιο θα συμβεί όποιος και αν είναι ο τύπος (εκτός από `char`) των συνιστωσών του πίνακα.

Φυσικά δεν είναι τυχαίο. Για τη C++:

- ***Ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.***

Στο Σχ. 12-1 ξαναδίνουμε το Σχ. 9-1 αναθεωρημένο σε συμφωνία με τα παραπάνω. Όπως καταλαβαίνεις, αν κάνεις αποπαραπομπή στο όνομα του πίνακα, θα πρέπει να πάρεις το x[0]:

```
cout << (*x) << endl;
```



Σχ. 12-1 »Διόρθωση» του Σχ. 9-1. Η σχηματική παράσταση πίνακα και των στοιχείων του με το βέλος προς το πρώτο στοιχείο (`pinA[0]`) του πίνακα.

Αποτέλεσμα:

```
0.1
```

Τι διαφορά έχει το βέλος `x` από το `p` που δηλώνουμε ως:

```
double* p;
```

Το `x` είναι ένα *σταθερό βέλος* ενώ η `p` είναι μια *μεταβλητή-βέλος*. Αν έχουμε δηλώσει:

```
double r1, r2;
```

μπορούμε να γράψουμε:

```
p = &r1;  p = &r2;
```

ενώ απαγορεύεται να αλλάξουμε την τιμή του βέλους `x`.

Με βάση αυτά, δες πώς μπορεί να γραφεί η *vectorSum()* που πρωτοείδαμε στην §9.3:

```
double vectorSum( const double* x, int n, int from, int upto )
{
   int m;
   double sum( 0 );

   for ( m = from; m <= upto; m = m + 1 )
      sum = sum + x[m];
   return sum;
} // vectorSum
```

Πρόσεξε ότι το μόνο που άλλαξε είναι η επικεφαλίδα, που ήταν:

```
double vectorSum( const double x[], int n, int from, int upto )
```

Στο σώμα της συνάρτησης δεν υπάρχει αλλαγή. Επίσης, δεν υπάρχει αλλαγή στη χρήση.

Παρατηρήσεις: ▶

1. Πάντως αξίζει να κάνουμε μερικές αλλαγές μέσα στο σώμα της συνάρτησης που δεν έχουν σχέση με το βέλος:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   double sum( 0 );

   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

2. Ο συμβολισμός με το βέλος δεν είναι κατ' ανάγκην προτιμότερος.

- Θα δεις στη συνέχεια ότι πολλές φορές θέλουμε να περάσουμε σε μια συνάρτηση (μέσω παραμέτρου) βέλος προς κάποιο αντικείμενο.
- Μέσα στη συνάρτηση δεν έχεις τρόπο να ξεχωρίσεις αν μια παράμετρος-βέλος δείχνει μια μεταβλητή ή έναν πίνακα.

Για λόγους τεκμηρίωσης λοιπόν, γενικώς,

♦ ***Όταν έχουμε παράμετρο-πίνακα θα προτιμούμε τον συμβολισμό με τις αγκύλες.***

και θα χρησιμοποιούμε παράμετρο-βέλος όταν περνούμε βέλος προς ένα αντικείμενο.

Ειδικώς για ορμαθούς χαρακτήρων της C –δηλαδή πίνακες χαρακτήρων με τον φρουρό `'\0'` στο τέλος– και οι δύο συμβολισμοί είναι καλοί αφού ναι μεν ο ορμαθός παριστάνεται με πίνακα αλλά θεωρείται ένα αντικείμενο.◀

Στην §9.3 γράφαμε ακόμη: «*Και αν θέλουμε το άθροισμα των πέντε τελευταίων στοιχείων* [*ενός πίνακα*]*; Θα μάθεις αργότερα έναν τρόπο να χρησιμοποιείς τη vectorSum()* [*χωρίς παραμέτρους περιοχής επεξεργασίας*] *και για την περίπτωση αυτή.*» Τώρα μπορούμε να το δούμε αυτό. Η απλή *vectorSum()* είναι:

```
double vectorSum( const double* x, int n )
{
```

```
    double sum( 0 );

    for ( int m(0); m < n; ++m ) sum += x[m];
    return sum;
} // vectorSum
```

Για τον πίνακα που δηλώσαμε πιο πάνω, τα πέντε τελευταία στοιχεία είναι τα x[4], x[5], x[6], x[7], x[8]. Αφού η συνάρτηση περιμένει το πρώτο όρισμα να δείχνει την αρχή του πίνακα και το δεύτερο το πλήθος των στοιχείων του την καλούμε ως εξής:

```
cout << vectorSum( &x[4], 5 ) << endl;
```

Δηλαδή, «ξεγελάμε» τη συνάρτηση περνώντας της για αρχή του πίνακα το στοιχείο `x[4]`.

Προσοχή όμως: Αυτό είναι ένα *τέχνασμα* που στηρίζεται στο ότι ξέρουμε πώς γίνεται η υλοποίηση των εννοιών στη C++ (και στη C)[1]. Το να περνούμε τον πίνακα (`const double x[], int n`) και την περιοχή επεξεργασίας (`int from, int upto`) είναι *πάγια τεχνική*.

12.2 Για τον Περιορισμό "const"

Λέγαμε στην προηγούμενη παράγραφο, μετά τη δήλωση "`double x[] = {...}`": «*Το* `x` *είναι ένα σταθερό βέλος* [...] *απαγορεύεται να αλλάξουμε την τιμή του βέλους* `x`.» Πράγματι, αν έχεις δηλώσει:

```
double r1;
```

η εντολή "`x = &r1`" δεν θα γίνει δεκτή από τον μεταγλωττιστή.

Παρ' όλο που τα διαγνωστικά είναι διαφορετικά, αυτό μας θυμίζει την περίπτωση που δηλώναμε (§2.4):

```
const double g( 9.81 ); // m/sec²
```

και –στη συνέχεια– ο μεταγλωττιστής δεν μας επιτρέπει τη "`g = 5.32`".

Πάντως, αν ξεχάσουμε την υλοποίηση και το βέλος και δεχθούμε ότι τιμή ενός πίνακα είναι οι τιμές όλων των στοιχείων του, τότε πιο κοντά σε αυτήν τη χρήση του "`const`" είναι αυτό που μάθαμε στην §9.3, στην παράμετρο-πίνακα μιας συνάρτησης:

```
double vectorSum( const double x[], int n, int from, int upto )
```

την οποία μπορούμε να γράψουμε και ως: "`const double* x`".

Με αυτά που μαθαίνουμε τώρα, βλέπουμε ότι, όταν πρόκειται για βέλος, υπάρχει και άλλη δυνατότητα: να αλλάζει ή να μην αλλάζει η τιμή του βέλους (δηλαδή η διεύθυνση της μνήμης που δείχνει). Μπορείς να επιβάλεις τη *σταθερότητα του βέλους* με την εξής δήλωση:

```
double* const x( &r1 );
```

Τελος, για να έχουμε *σταθερό βέλος προς σταθερή τιμή* θα πρέπει να δηλώσουμε:

```
const double* const x( &g );
```

Τέτοιες δηλώσεις θα βρεις συνήθως στις τυπικές παραμέτρους συναρτήσεων. Εκεί φυσικά δεν βάζουμε αρχική τιμή· αυτή καθορίζεται αυτομάτως όταν καλείται η συνάρτηση και είναι η αντίστοιχη πραγματική παράμετρος.

12.2.1 Τυποθεώρηση "const"

Ας ξαναγυρίσουμε στην §9.5.1. Στη συνάρτηση *linSearch()* (με φρουρό) υποχρεωθήκαμε να αλλάξουμε το "`const int v[]`" σε "`int v[]`" διότι ο μεταγλωττιστής δεν δεχόταν τις εντολές:

```
        v[upto+1] = x;       // φρουρός
    // . . .
```

[1] ... και –καλώς ή κακώς– θα το δεις σε πολλά βιβλία, σε πολλούς διαδικτυακούς τόπους κλπ.

```
        v[upto+1] = save;    // όπως ήταν στην αρχή
```

Αν σκεφτούμε ότι στην πραγματικότητα "`const int v[]`" σημαίνει "`const int* v`" μπορούμε να κάνουμε το εξής: Ορίζουμε μια μεταβλητή

```
        int* ncv( v );
```

Η *ncv* είναι βέλος προς `int`, όπως και η *v*, αλλά δεν είναι `const`. Βάζοντας την *ncv* να δείχνει όπου και η *v* θα μπορούμε να τροποιούμε τα στοιχεία του πίνακα *v* μέσω αυτής. Αυτό όμως δεν μπορεί να γίνει ή τουλάχιστον δεν μπορεί να γίνει έτσι· δεν το επιτρέπει ο μεταγλωττιστής. Μπορεί να γίνει με τη λεγόμενη **τυποθεώρηση const**:

```
        int* ncv( const_cast<int*>(v) );
```

δηλαδή: το ncv δείχνει τον ίδιο πίνακα που δείχνει και το v, αλλά, αφού δεν έχει "`const`" (`const_cast<int*>`) μας δίνει συνατότητα τροποποίησης.

Να η νέα μορφή της *linSearch()*:

```
 0: int linSearch( const int v[], int n,
 1:                int from, int upto, int x )
 2: {
 3:    int fv( -1 );
 4:
 5:    if ( 0 <= from && from <= upto && upto < n )
 6:    {
 7:       int* ncv( const_cast<int*>(v) );
 8:       int save( v[upto+1] ); // φύλαξε το v[upto+1]
 9:       ncv[upto+1] = x;       // φρουρός
10:       int k( from );
11:       while ( v[k] != x )  ++k;
12:       if ( k <= upto )
13:          fv = k;
14:       ncv[upto+1] = save;    // όπως ήταν στην αρχή
15:    // (from <= fv <= upto && v[fv] == x) ||
16:    //       (fv == -1 && (για κάθε j:from..upto • v[j] != x))
17:    }
18:    return fv;
19: } // linSearch
```

Πρόσεξε τώρα τα εξής:

- Στην επικεφαλίδα (γρ. 0) *δεν αλλάξαμε* το "`int v[]`" σε "`int* v`". Όπως είπαμε, το πρώτο είναι προτιμότερο μια και δείχνει πίνακα.
- Μεταφέραμε τις δηλώσεις των *save* (γρ. 8) και *k* (γρ. 10) εκεί που μπορούμε να ορίσουμε και την αρχική τους τιμή.
- Στη γρ. 7 βάλαμε τη δήλωση του *ncv* με την τυποθεώρηση `const`.
- Έτσι, όταν στη γρ. 9 και στη γρ. 14 αλλάζουμε την τιμή του ncv`[`*upto*+1`]` αλλάζουμε στην πραγματικότητα την τιμή του v`[`*upto*+1`]`.

Με την τυποθεώρηση **const** μπορείς να αφαιρείς ή να προσθέτεις τον περιορισμό "`const`" (ή τον "`volatile`"). Για παράδειγμα, ας πούμε ότι θέλουμε να έχουμε έναν πίνακα που οι τιμές των στοιχείων του δεν θα αλλάζουν κατά τη διάρκεια εκτέλεσης του προγράμματος. Αλλά, αυτές οι τιμές δεν είναι σταθερές γνωστές όταν γράφουμε το πρόγραμμα. Θα πρέπει κάθε φορά που εκτελείται να διαβάζονται από κάποιο αρχείο. Μπορούμε να κάνουμε το εξής:

```
   double xf[100];
// ανάγνωση τιμών του xf από το αρχείο
   const double* x( const_cast<const double*>(xf) );
```

Τώρα, μέσω του x, μπορούμε να χρησιμοποιούμε τα στοιχεία χωρίς να υπάρχει περίπτωση να αλλάξουμε κάποια τιμή κατά λάθος. Δυστυχώς όμως και το xf υπάρχει και η ζημιά μπορεί να γίνει από εκεί.

Να ένας άλλος τρόπος πιο ασφαλής:

```
   const double x[100] = { 0 };
   double* xf( const_cast<double*>(x) );
```

```
// ανάγνωση τιμών του xf από το αρχείο
   xf = 0;
```

Εδώ χρησιμοποιούμε το βέλος xf για να δώσουμε τιμές στα στοιχεία του x, που είναι δηλωμένος `const`. Μετά από αυτό, του βάζουμε τιμή 0 και τον «αποσυνδέουμε» από τον πίνακα x. Για την τιμή βέλους 0 τα λέμε παρακάτω.

Αργότερα θα δούμε παράδειγμα τυποθεώρησης **const** και σε παράμετρο αναφοράς

Τα ίδια μπορείς να κάνεις και με το "`volatile`" αλλά κάτι τέτοιο δεν είναι και τόσο χρήσιμο.

12.3 Πράξεις με Βέλη

Τώρα μπορούμε να δούμε και κάτι άλλο: Με τα βέλη προς στοιχεία πίνακα μπορείς να κάνεις και πράξεις, για την ακρίβεια πρόσθεση και αφαίρεση. Έτσι, έχουν νόημα τα `x+1`, `x+2`,... που μας δίνουν τις θέσεις των στοιχείων `x[1]`, `x[2]`,... Με αποπαραπομπή (`*(x+1)`, `*(x+2)`,...) παίρνουμε τα στοιχεία:

```
   cout << (*x) << "  " << (*(x+1)) << "  " << (*(x+2)) << endl;
```

Αποτέλεσμα:

```
0.1  1.2  2.3
```

Εδώ πρόσεξε το εξής: Οι παρενθέσεις *είναι απαραίτητες* διότι *η πράξη της αποπαραπομπής έχει μεγαλύτερη προτεραιότητα από την πρόσθεση*. Πράγματι από την

```
   cout << (*x+1) << "  " << (*(x+1)) << endl;
```

θα πάρουμε:

```
1.1  1.2
```

Το `1.1` είναι στην πραγματικότητα το αποτέλεσμα της πράξης: `*x+1` που δεν είναι τίποτε άλλο από το `x[0]+1`.

Να λοιπόν η σχέση μεταξύ *δείκτη στοιχείου* και *βέλους προς στοιχείο* πίνακα: Αν

```
T x[N];
```

και *k* φυσικός από 0 μέχρι *N*-1, τότε:

```
x[k] == *(x + k)
```

Πρόσεξε το εξής: το `*(x+k)` δεν είναι η τιμή του στοιχείου αλλά το ίδιο το στοιχείο (δηλαδή μια *τιμή-l*). Αυτό σημαίνει ότι μπορείς να δώσεις:

```
   *(x+3) = 7.77;
```

για να αλλάξεις την τιμή του `x[3]`.

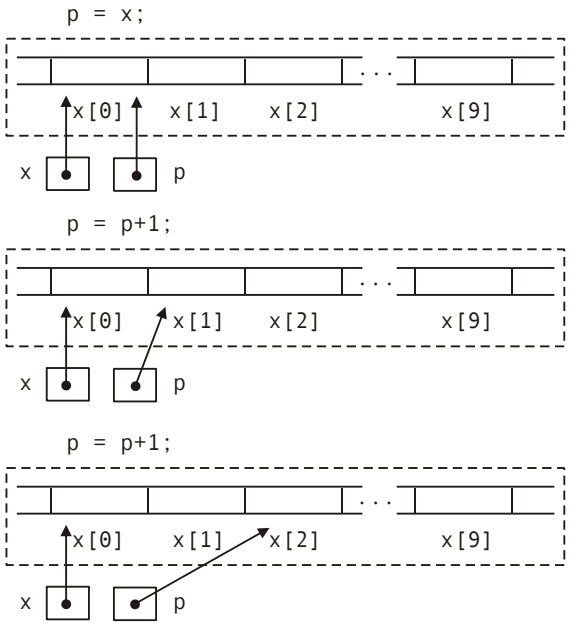
Είδαμε πιο πριν ότι μπορούμε να υπολογίσουμε το άθροισμα των στοιχείων ενός πίνακα με τις:

```
      double sum( 0 );
      for ( int m(0); m < N; ++m ) sum += x[m]; // άθροισμα
```

Μπορούμε να κάνουμε τον ίδιο υπολογισμό με βέλη. Ας πούμε ότι έχουμε μια μεταβλητή-βέλος *p* προς μεταβλητή τύπου `double`. Μπορούμε να δώσουμε τιμή στην *p* με την εντολή: `p = x`. Με την `*p` παίρνουμε την τιμή του x[0]. Αυξάνοντας την τιμή της *p* κατά 1 (`p=p+1` ή `++p`) βάζουμε την *p* να δείχνει το x[1] και με αποπαραπομπή (`*p`) παίρνουμε την τιμή του. Δες το Σχ. 12-2. Για να πάρουμε λοιπόν το άθροισμα μπορούμε να γράψουμε:

```
      double sum( 0 );
      for ( double* p(x); p != &x[N]; ++p )  sum += *p;
```

Δηλαδή: Θέλω μια μεταβλητή *p* τέτοια ώστε το `*p` να είναι τύπου `double`. Αρχικώς θα δείχνει το στοιχείο `x[0]` του πίνακα x (`p(x)`). Στη συνέχεια θα πηγαίνει στο επόμενο στοιχείο (`++p`) και θα σταματήσει όταν βρεθεί να δείχνει την πρώτη θέση μετά το τελευταίο στοιχείο του πίνακα (`&x[N]`). Αυτή η δήλωση και η τελευταία **for** εξετάζονται πιο εκτεταμένα στην επόμενη υποπαράγραφο.



Σχ. 12-2 Ο πίνακας και τα βέλη. Τιμή του πίνακα (`x`) είναι η διεύθυνση του πρώτου στοιχείου (x[0]). Εδώ, βάζουμε ένα βέλος που δείχνει το x[0], αντί να βάλουμε ως τιμή της x την `0x0064fdbc` (ή κάτι παρόμοιο),. Μετά την εκτέλεση της εκχώρησης "`p = x`" την ίδια τιμή έχει και η *p*. Όταν αυξήσουμε την τιμή της *p* κατά 1 (`p=p+1`) η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[1]). Αν αυξήσουμε ξανά την τιμή της *p* κατά 1 η *p* δείχνει το επόμενο στοιχείο του πίνακα (x[2]) κ.ο.κ..

Πολλά πράγματα μαζί! Ας τα δούμε ένα-ένα.

12.3.1 Μια Θέση Μετά το Τέλος

Πριν από οποιαδήποτε πράξη να πούμε το εξής: Στο παραπάνω παράδειγμα εμφανίστηκε ένα "`&x[N]`". Τι είναι αυτό; Τα στοιχεία του πίνακα καταλαμβάνουν τις θέσεις από 0 μέχρι *N*-1. Σωστό! Αλλά η C++ σου επιτρέπει να έχεις βέλος που να δείχνει στο `x[N]` (past the end pointer), αρκεί:

- Να μήν προσπαθήσεις να κάνεις αποπαραπομπή.
- Να το χρησιμοποιείς μόνο σε συγκρίσεις με άλλα βέλη (π.χ.: "`p != &x[N]`")

12.3.2 Αρχική Τιμή και Εκχώρηση

Ας ξεκινήσουμε από τον ορισμό αρχικής τιμής "`double* p(x)`". Σε μια μεταβλητή-βέλος μπορείς να δίνεις ως αρχική τιμή την τιμή μιας παράστασης-βέλος του ίδιου τύπου. Από όσα ξέρουμε μέχρι τώρα μπορείς να δώσεις:

- μια άλλη μεταβλητή-βέλος ίδιου τύπου· αν *p*1 τύπου `double*`: "`double* p(p1)`", οπότε το *p* δείχνει την ίδια θέση της μνήμης που δείχνει και το *p*1 (*p* == *p*1 ή `*`*p* και `*`*p*1 είναι το ίδιο αντικείμενο)
- το όνομα ενός πίνακα, όπως εδώ: "`double* p(x)` ", οπότε το *p* δείχνει το πρώτο στοιχείο του πίνακα (*p* == &x[0])
- τη διεύθυνση μιας μεταβλητής, π.χ.: "`double* p(&sum)`", οπότε το *p* δείχνει τη μεταβλητή *sum* (*p* == &*sum* ή `*`*p* είναι η *sum*).

  Όπως θα δούμε στη συνέχεια, μπορείς ακόμη να δώσεις:
- τιμή μηδέν (0): "`double* p( 0 )`",

- την τιμή αριθμητικής πράξης με βέλη, π.χ.: "`double* p(x+1)`".

Προσοχή! ▶

Σε ένα βέλος τύπου `const T*` μπορείς να βάλεις ως αρχική τιμή βέλος τύπου `T*`, π.χ.:

```
int a[17];
const int* pa( a );
```

Το αντίθετο δεν επιτρέπεται. Μπορείς να το πετύχεις με την κατάλληλη τυποθεώρηση, π.χ.:

```
unsigned int myStrLen( const char* cs )
{
   char* p( const_cast<char*>(cs) );
// . . .
```

(από ένα παράδειγμα που θα δεις στη συνέχεια.) ◀

Όταν δηλώνεις ένα βέλος, χωρίς να του δώσεις αρχική τιμή αυτό δείχνει σε κάποια τυχαία θέση μέσα στη μνήμη. Αν προσπαθήσεις να το χρησιμοποιήσεις μπορεί να κάνεις και κάποια ζημιά. Γι' αυτό οι προγραμματιστές συνηθίζουν να δίνουν αρχική τιμή στα βέλη, με τη δήλωσή τους. Αν δεν ξέρουν τι τιμή να δώσουν δίνουν την τιμή μηδέν ("`0`")[2] που:

- είναι συμβατή με οποιονδήποτε τύπο-βέλους,
- σημαίνει (κατά κοινή συμφωνία): *αυτό το βέλος δεν δείχνει κάτι* (που μας ενδιαφέρει) και
- είναι τιμή που μπορεί να ελεγχθεί (`if (p == 0)...`), όπως θα δούμε στη συνέχεια.

Με τους ίδιους τρόπους μπορείς να αλλάζεις, στη συνέχεια, την τιμή ενός βέλους με μια εντολή εκχώρησης. Στο δεξιό μερος μπορεί να υπάρχει "`0`" ή παράσταση που θα μας δώσει τιμή-βέλος ίδιου τύπου. Για παράδειγμα:

```
p = p1;
p = x;
p = &sum;
p = 0;
p = x + 1;
```

Θυμίσου, για παράδειγμα, ότι στην §12.2, αποσυνδέσαμε το βέλος *xf* από τον πίνακα x βάζοντας "`xf = 0`".

Για τον περιορισμό "`const`", ισχύει και για την εκχώρηση ο περιορισμός που είπαμε πιο πάνω για την απόδοση αρχικής τιμής.

Ένα σημείο που χρειάζεται προσοχή είναι το εξής: Ας πούμε ότι έχουμε δηλώσει:

```
int a( 10 ), b( 20 );
int* pa;
int* pb;
```

και στη συνέχεια δίνουμε:

```
pa = &a;  pb = &b;
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 12-3(α): το βέλος *pa* δείχνει την *a*, που έχει τιμή 10, και το βέλος *pb* δείχνει την *b* που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή "`a = b`". Η εικόνα της μνήμης είναι αυτή του Σχ. 12-3 (β). Η τιμή της *a* γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

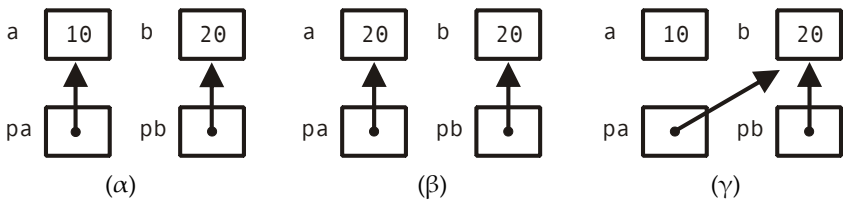
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: "`pa = pb`". Οι τιμές των μεταβλητών *a* και *b* δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *pa*: το βέλος *pa* δείχνει εκεί που δείχνει και το *pb*: τη *b* (Σχ. 12-3 (γ)).

Φυσικά, και στις δύο περιπτώσεις, αν δώσουμε την εντολή:

```
cout << *pa << "  " << *pb << endl;
```

---

[2] Οι προερχόμενοι από τη C αντί "`0`" (μηδέν) βάζουν τη σταθερά "`NULL`" που ορίζεται με τη μάκρο "`#define NULL 0`". Ο B. Stroustrup συμβουλεύει να προτιμούμε το "`0`". Πάντως στη νέα τυποποίηση C++11 εισάγεται το όνομα (*std*`::`)"*nullptr*".



Σχ. 12-3 Αρχικώς, μετά την εκτέλεση των εντολών "`pa = &a; pb = &b;`" έχουμε την εικόνα (α). Η εικόνα (β) προκύπτει αν στη συνέχεια εκτελεσθεί η εκχώρηση: "`a = b`". Η εικόνα (γ) προκύπτει αν, μετά την (α), εκτελεσθεί η εκχώρηση "`pa = pb`".

θα πάρουμε αποτέλεσμα:

```
20  20
```

Υπάρχει και ένας περιορισμός σχετικά με την εκχώρηση: *η μεταβλητή και η παράσταση θα πρέπει να είναι του ίδιου τύπου*. Έτσι, αν έχουμε δηλώσει:

```
double v;
```

δεν επιτρέπεται να γράψουμε[3]: "`pa = &v`".

Προσοχή! ▶

Πριν προχωρήσουμε να επισημάνουμε κάτι που μπορεί να σε οδηγήσει σε «παράξενα» λάθη μεταγλώττισης. Αν θελήσεις να μαζέψεις τις δύο δηλώσεις:

```
int* pa;
int* pb;
```

σε μία, θα πρέπει να γράψεις:

```
int *pa, *pb;
```

Αν γράψεις κατά λάθος:

```
int* pa, pb;
```

θα σημαίνει ότι η *pa* είναι βέλος προς `int` αλλά η *pb* είναι `int`! ◀

12.3.3 Πρόσθεση και Αφαίρεση

Αν έχεις δηλώσει:

```
const unsigned int N = ...;
T x[N];
```

ορίζονται οι πράξεις: `+`: *T*`*`×`int` ↠ *T*`*` και `-`: *T*`*`×`int` ↠ *T*`*` ή, ακριβέστερα:

`+`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`+`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`&x[0]` .. `&x[`*N*`]`] × [`-`*N* .. *N*] ↠ [`&x[0]` .. `&x[`*N*`]`]

`-`: [`-`*N* .. *N*] × [`&x[0]` .. `&x[`*N*`]`] ↠ [`&x[0]` .. `&x[`*N*`]`]

Πιό πάνω είδαμε ότι αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] `..` x[*N*`-1`], τότε *το* `p+1` *δείχνει το επόμενο στοιχείο*. Αντιστοίχως, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[1] .. x[*N*], τότε *το "*`p-1`*" δείχνει το προηγούμενο στοιχείο*.

Γενικώς, αν το βέλος *p* δείχνει κάποιο από τα στοιχεία x[0] .. x[*N*`-1`], τότε

- *Το "*`p + k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***μετά*** *το στοιχείο που δείχνει το p*.
- *Το "*`p - k`*"* (*k*: *φυσικός αριθμός*) *δείχνει το στοιχείο του πίνακα που βρίσκεται k θέσεις* ***πριν*** *το στοιχείο που δείχνει το p*.

[3] Αν επιμένεις πολύ γίνεται, με την κατάλληλη τυποθεώρηση· αλλά, πού θα σου χρησιμεύσει κάτι τέτοιο (εκτός από το "σκαλίζεις" τη μνήμη);

Προσοχή! ▶

1. Δεν ειναι τυχαίες οι συνεχείς αναφορές σε πίνακες. *Οι πράξεις αυτές ορίζονται μόνο για βέλη προς στοιχεία ενός πίνακα.*

2. Παρ' όλο που λέμε ότι «τιμή του βέλους *p* είναι μια διεύθυνση της μνήμης» *το* "`p + 1`" ***δεν*** *είναι η επόμενη διεύθυνση* (π.χ. η διεύθυνση της επόμενης ψηφιολέξης). Είναι η διεύθυνση του επόμενου στοιχείου του πίνακα.◀

Γιατί γράψαμε τις συναρτήσεις μας μερικές; Διότι, για παράδειγμα, αν το *p* δείχνει το x[2] απαγορεύονται πράξεις σαν τις `p-6` ή `p+(-5)`.

- ***Είναι υποχρέωση του προγραμματιστή να φροντίσει ώστε το αποτέλεσμα πράξης βελών να είναι βέλος προς στοιχείο πίνακα ή προς την πρώτη θέση μετά το τελευταίο στοιχείο.***

Οι συντομογραφίες "`++`", "`--`", "`+=`", "`-=`" ισχύουν και έχουν το νόημα που ξέρεις προσαρμοσμένο στα παραπάνω. Έτσι, αν το *p* δείχνει το x[2] το "`++p`" αλλάζει την τιμή του *p* ώστε να δείχνει το x[3] (ενώ αντιθέτως το "`--p`" θα το πήγαινε στο x[1]). Αν βάλεις "`p += 2`", το *p* θα πάει από το x[2] στο x[4].

Υπάρχει μια ακόμη συνάρτηση (πράξη)· είναι μεταξύ βελών και είναι ολική:

`-`: [`&x[0] .. &x[`*N*`]`] × [`&x[0] .. &x[`*N*`]`] → [`-`*N* `..` *N*]

Αν δύο βέλη, `p1`, `p2`, δείχνουν στοιχεία του ίδιου πίνακα, ας πούμε τα `x[k1]` και `x[k2]`, έχει νόημα η διαφορά "`p1-p2`", που δείχνει τον αριθμό των στοιχείων που πρέπει να διανύσουμε για να φτάσουμε από το ένα στοιχείο στο άλλο· στην περίπτωσή μας "`k1-k2`".

Ας δούμε δύο παραδείγματα που χρησιμοποιούν πράξεις μεταξύ βελών.

Παράδειγμα 1 – Μήκος ορμαθού C

Όπως μάθαμε στο Κεφ. 10 (§10.13), στη C παριστάνουμε τα κείμενα σε πίνακες με στοιχεία τύπου `char` που έχουν στο τέλος (ως φρουρό) τον χαρακτήρα `'\0'`. Μια από τις συναρτήσεις που μας δίνει η C για τον χειρισμό τέτοιων πινάκων είναι η *strlen()* που μας δίνει το μήκος του κειμένου. Εδώ θα γράψουμε μια τέτοια συνάρτηση, ας την πούμε *myStrLen()*.

Αφού παρατηρήσουμε ότι αν *l* το μήκος ενός κειμένου αποθηκευμένου στον πίνακα cs τοτε ο φρουρός βρίσκεται στο στοιχείο cs`[`*l*`]`, γράφουμε τη συνάρτησή μας ως εξής:

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l] != '\0' )  ++l;
   return l;
} // myStrLen
```

Γράφουμε τώρα το ίδιο πράγμα με βέλη, για να πάρουμε μια πιο γρήγορη συνάρτηση:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *p != '\0' )  ++p;
   return p-cs;
} // myStrLen
```

Το *p* θα διατρέξει ολόκληρον τον πίνακα ξεκινώντας από το `cs[0]` (διεύθυνση `&cs[0]` ή απλώς `cs`). Ναι, αλλά πώς θα γίνει αυτό; Έχουμε βάλει "`const char* p(cs)`"! Αν έχεις αυτήν την απορία γύρισε πίσω, στην §12.2. Αυτό που λέμε εδώ είναι ότι *δεν επιτρέπεται μεταβολή αυτού που δείχνει το βέλος· μεταβολή του βέλους επιτρέπεται.*

Αν όμως σε ενοχλεί το "`const`" πρόσεξε τα εξής: Ο μεταγλωττιστής δεν θα επιτρέψει να γράψουμε `char* p(cs)` αφού το cs είναι "`const char`". Η τυποθεώρηση **const** είναι απαραίτητη:

```
   char* p( const_cast<char*>(cs) );
```

Ο *p* προχωρεί με τη `++p` όσο αυτό που δείχνει δεν έχει τον φρουρό (`*p != '\0'`).

Τελικώς, η συνάρτηση επιστρέφει τη διαφορά του βέλους προς τον φρουρό από το βέλος προς την αρχή.

Ένας προγραμματιστής C θα απορούσε ήδη με το ότι στην πρώτη μορφή επιμένουμε να γράφουμε πολλά αντί να χρησιμοποιήσουμε τον μεταθεματικό "`++`":

```
size_t myStrLen( const char cs[] )
{
   unsigned int l( 0 );
   while ( cs[l++] != '\0' );
   return l-1;
} // myStrLen
```

Πρόσεξε ότι εδώ παίρνουμε την τιμή του *l* για να υπολογίσουμε το στοιχείο cs`[`*l*`]` και αυξάνουμε την τιμή του *l*. Έτσι, αφ'ενός δεν έχουμε (άλλη) επαναλαμβανόμενη εντολή στη **while**, αφ' ετέρου η τιμή του *l* θα αυξηθεί και στην τελευταία επανάληψη, όταν βρούμε τον φρουρό. Γι' αυτό και επιστρέφουμε *l*-1.

Να το γράψουμε με βέλη; Νάτο:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) != '\0' );
   return p-cs-1;
} // myStrLen
```

Πρόσεξε ότι χρησιμοποιούμε μεταθεματικό "`++`" στο βέλος. Φυσικά, θα πρέπει και εδώ να αφαιρέσουμε 1.

Και βέβαια, ένας προγραμματιστής C δεν θα έλεγχε ποτέ για μηδέν με αυτόν τον τρόπο· θα το έγραφε πολύ πιο απλά:

```
size_t myStrLen( const char cs[] )
{
   const char* p( cs );
   while ( *(p++) );
   return p-cs-1;
} // myStrLen
```

🖘🖘🖘

Παράδειγμα 2 – Επισύναψη ορμαθού C 🖙

Στην §10.13 λέγαμε και για τη *strcat*: «*Η* `strcat(a, b)` *κάνει το ίδιο πράγμα με την* `a.append(b)`*: επισυνάπτει στο τέλος της τιμής του a την τιμή του b.*» Η επικεφαλίδα της είναι:

```
char* strcat( char* dest, const char* src )
```

Η *strcat*, αφού επισυνάψει στο τέλος του *dest* ένα αντίγραφο του *src*, επιστρέφει ως τιμή βέλος προς την αρχή του *dest*.

Ας υλοποιήσουμε μια τέτοια *myStrCat* με πίνακες και δείκτες· χωρίς βέλη:

```
char* myStrCat( char* dest, const char* src )
{
   int j( 0 );
   while ( dest[j] != '\0' ) ++j;
   int k(0);  // εδώ έχουμε: dest[j] == '\0'
   while ( src[k] != '\0' )
   {
      dest[j] = src[k];
      ++j;  ++k;
   } // for
   dest[j] = '\0';
   return dest;
} // myStrCat
```

Πρόσεξε ότι δεν έχουμε βάλει οποιονδήποτε έλεγχο. Σκέψου, ως άσκηση, τους ελέγχους που θα πρέπει να βάλουμε.

Η ίδια συνάρτηση με βέλη μπορεί να γραφεί ως εξής:

```
char* myStrCat( char* dest, const char* src )
{
   char* pd( dest );
   while ( *(pd++) != '\0' );
   --pd;       // εδώ έχουμε το pd να δείχνει τον φρουρό
   const char* ps( src );
   while ( *ps != '\0' )
   {
      *pd = *ps;
      ++pd;  ++ps;
   }
   *pd = '\0';
   return dest;
} // myStrCat
```

Γιατί "`--pd`"; Ξαναδιάβασε το προηγούμενο παράδειγμα...

12.3.4 Ο Τύπος "ptrdiff_t"

Στο Παράδ. 1 της προηγούμενης παραγράφου, σε συμμόρφωση με τη φιλοσοφία της C++, βάλαμε ως τύπο επιστροφής "`size_t`".

Εδώ θα πρέπει να αναφέρουμε ότι το αποτέλεσμα της πράξης

`-`: [`&x[0] .. &x[`N`]`] × [`&x[0] .. &x[`N`]`] → [`-`N `..` N]

είναι τύπου "`std::ptrdiff_t`" (*poin*ter *diff*erence) . Αν ψάξεις στο `cstddef` (ή το `stddef.h`) θα βρεις:

```
typedef int     ptrdiff_t;
```

ή κάτι παρόμοιο.

Όταν γράφουμε "`return p-cs`" ζητούμε να επιστραφεί μια τιμή τύπου `ptrdiff_t`. Επειδή όμως αυτή είναι σίγουρα μη αρνητική μετατρέπεται σε τύπο `size_t`.

12.3.5 Συγκρίσεις

Ένα άλλο πράγμα που μπορείς να κάνεις με τα βέλη είναι η σύγκριση για ισότητα ("`==`") ή για ανισότητα ("`!=`"). Έτσι, βλέπεις στη **for** της άθροισης τη σύγκριση: "`p != x+N`" ή "`p != &x[N]`".

Μπορείς να συγκρίνεις δύο βέλη του ίδιου τύπου. Αργότερα θα δούμε ότι μερικές φορές χρειάζεται να συγκρίνουμε και βέλη διαφορετικού τύπου. Αυτό γίνεται με την κατάλληλη τυποθεώρηση.

Παράδειγμα

Στη *vectorSum* δεν έχουμε βάλει ελέγχους. Έτσι, μπορεί να κληθεί με $n \le 0$, με *from* ή/και *upto* έξω από την περιοχή 0..n-1. Αυτά σου τα αφήνουμε ως άσκηση.

Εδώ θα δούμε ένα άλλο πρόβλημα: Όπως καταλαβαίνεις, η *vectorSum* μπορεί να κληθεί με πρώτη παράμετρο 0 (μηδέν), δηλαδή *χωρίς πίνακα*. Και αν μεν έχουμε και n `==` 0 μπορούμε να βγάλουμε μηδενικό άθροισμα. Αν όμως έχουμε n `>` 0 τότε έχουμε προφανώς λάθος. Να λοιπόν η συνάρτηση με αυτόν τον έλεγχο:

```
double vectorSum( const double x[], int n, int from, int upto )
{
   if ( x == 0 && n > 0 )
   {  cerr << "η vectorSum κλήθηκε με ανύπαρκτο πίνακα" << endl;
      exit( EXIT_FAILURE );  }
// άλλοι έλεγχοι
   double sum( 0 );
   for ( int m(from); m <= upto; ++m )  sum += x[m];
   return sum;
} // vectorSum
```

12.4 Πολυδιάστατοι Πίνακες

Ας πούμε ότι έχουμε ορίσει:

```
typedef int   intArr [ 51 ];
```

και δηλώνουμε:

```
   intArr    mat [ 11 ];
```

Τι είναι μια συνιστώσα του mat, π.χ. η `mat[7]`; Ένας *πίνακας* με 51 συνιστώσες τύπου `int`. Πώς θα αναφερθούμε στην 23η συνιστώσα αυτού του πίνακα; Αν θεωρήσουμε ότι έχουμε έναν μονοδιάστατο πίνακα με όνομα `mat[7]` το 23ο στοιχείο του θα είναι: `(mat[7])[23]`. Η C++ μας επιτρέπει να γράψουμε απλούστερα: `mat[7][23]`.

Η δήλωση του mat θα μπορούσε να δοθεί ισοδυνάμως και ως εξής:

```
   int   mat[ 11 ][ 51 ];
```

Ο mat είναι ένας **πίνακας δυο διαστάσεων** (two-dimensional array) και έχει 11×51 στοιχεία τύπου `int`.

Όταν δηλώνουμε έναν πίνακα, μονοδιάστατο ή πολυδιάστατο, ο τύπος συνιστωσών μπορεί να είναι οποιοσδήποτε. Έτσι, οι πίνακες pinA, pinB, pinC και pinD, που δηλώνονται στο παρακάτω παράδειγμα, είναι πίνακες μιας διάστασης, δύο, τριών και τεσσάρων διαστάσεων αντίστοιχα.

```
   int    pinA[ 10 ];
   char   pinB[ 10 ][ 30 ];
   double pinC[ 5 ][ 10 ][ 20 ];
   bool   pinD[ 5 ][ 10 ][ 20 ][ 8 ];
```

- Ο πίνακας pinA αποτελείται από 10 στοιχεία τύπου `int`,
- ο πίνακας pinB από 300 (= 10×30) στοιχεία τύπου `char`,
- ο πίνακας pinC από 1000 (= 5×10×20) στοιχεία τύπου `double` και τέλος
- ο πίνακας pinD από 8000 (= 5×10×20×8) στοιχεία τύπου `bool`.

Να μερικά παραδείγματα γραφής στοιχείων πολυδιάστατων πινάκων, σύμφωνα με τις παραπάνω δηλώσεις:

```
α) pinB[1][1], pinB[1][29], pinB[2][1], pinB[2][29],
   pinB[9][1], pinB[9][29]
β) pinC[1][1][1], pinC[5][1][1], pinC[1][9][1],
   pinC[1][1][19], pinC[5][9][19]
γ) pinD[1][1][1][1], pinD[1][1][1][2], pinD[4][9][19][7],
   pinD[4][9][19][8]
```

Με τη δήλωση μπορείς να δώσεις και αρχική τιμή. Αφού, όπως είπαμε, ένας δισδιάστατος πίνακας είναι (μονοδιάστατος) πίνακας μονοδιάστατων πινάκων θα πρέπει να μπορούμε να δώσουμε κάτι σαν:

```
   double x[4][2] = { {0.1, 1.2}, {2.3, 3.4}, {4.5, 5.6},
                      {6.7, 7.8} };
```

Και πράγματι, αυτό είναι σωστό! Και ποια είναι τα στοιχεία αυτού του μονοδιάστατου πίνακα; Τα `x[0]`,… `x[3]`. Οι εντολές:

```
   for ( int r(0); r < 4; ++r )
      cout << x[r] << "  " << &x[r][0] << endl;
```

θα μας δώσουν:

```
0x22ff30  0x22ff30
0x22ff40  0x22ff40
0x22ff50  0x22ff50
0x22ff60  0x22ff60
```

Δηλαδή, το `x[r]` είναι βέλος προς το `x[r][0]`.

Στη C++ τα στοιχεία ενός δισδιάστατου πίνακα *αποθηκεύονται κατά γραμμές*: πρώτα τα στοιχεία της γραμμής 0, στη συνέχεια τα στοιχεία της γραμμής 1 κλπ. Αυτό όμως δεν σημαίνει ότι έχουμε την υποχρέωση να επεξεργαζόμαστε τα στοιχεία με κάποια συγκεκριμένη σειρά. *Δεν υπάρχει κανένας περιορισμός σχετικά με τη σειρά επεξεργασίας των στοιχείων του πίνακα. Η σειρά αυτή καθορίζεται μόνο από τις ανάγκες και τη λογική του προγράμματος και επιλέγεται ελεύθερα από τον προγραμματιστή.*

Ας δούμε μερικά παραδείγματα:

Παράδειγμα 1 - Άθροισμα στοιχείων τρισδιάστατου πίνακα

Αν έχουμε δηλώσει:

```
double pinC[ 5 ][ 10 ][ 20 ];
```

οι τρεις παρακάτω φωλιασμένες **for** υπολογίζουν το άθροισμα, *sum*, των στοιχείων του pinC:

```
sum = 0;
for ( int k(0); k < 5; ++k ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int m(0); m < 20; ++m ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int m . . .
   } // for ( int l . . .
} // for ( int k . . .
```

Όπως καταλαβαίνεις, ο τρίτος δείκτης, δηλαδή ο *m*, είναι ο δείκτης που αλλάζει τιμή πιο συχνά από τους άλλους και αυτό επειδή βρίσκεται στην πιο εσωτερική **for**. Ο δείκτης *m* διατρέχει όλες τις τιμές του, δηλ.τις τιμές 0 μέχρι 19, για κάθε νέο συνδυασμό τιμών των δεικτών *k* και *l* (δηλ. για 50 συνδυασμούς). Έτσι, η εσωτερική **for** εκτελείται 1000 (=20×50) φορές. Αντίθετα, ο δείκτης *k* είναι ο δείκτης που *μόνο μια φορά* διατρέχει τις τιμές του (από 0 ως 4), ενώ ο δείκτης *l* διατρέχει 5 φορές τις τιμές του (από 0 μέχρι 9). Έτσι, η ενδιάμεση **for** εκτελείται 5 φορές.

Θα μπορούσαμε να υπολογίσουμε το άθροισμα και με τις:

```
sum = 0;
for ( int m(0); m < 20; ++m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(0); l < 10; ++l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(0); k < 5; ++k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
} // for ( int m . . .
```

Στη περίπτωση αυτή, ο πρώτος δείκτης, *k*, είναι αυτός που αλλάζει τιμή πιο συχνά και διατρέχει 200 φορές τις τιμές του, δηλ. τις τιμές 0 μέχρι 4. Ο δείκτης *m* διατρέχει μόνο μιά φορά τις τιμές του (0 ως 19) και ο *l* διατρέχει 20 φορές τις τιμές του (από 0 μέχρι 9).

Τέλος, δες και μια τρίτη περίπτωση:

```
sum = 0;
for ( int m(19); m >= 0; --m ) // ΕΞΩΤΕΡΙΚΗ
{
   for ( int l(9); l >= 0; --l ) // ΕΝΔΙΑΜΕΣΗ
   {
      for ( int k(4); k >= 0; --k ) // ΕΣΩΤΕΡΙΚΗ
      {
         sum += pinC[k][l][m];
      } // for ( int k . . .
   } // for ( int l . . .
```

```
   } // for ( int m . . .
```

Στην περίπτωση αυτήν έχουμε το ίδιο φώλιασμα με τη δεύτερη, αλλά οι δείκτες διατρέχουν τις τιμές από το τέλος προς την αρχή.

Παράδειγμα 2 - Ανάγνωση τιμών στοιχείων δισδιάστατου πίνακα

Ας δούμε όμως και το εξής πρόβλημα: Σε ένα αρχείο, που το διαβάζουμε μέσω του ρεύματος *ins*, έχουμε τις τιμές των στοιχείων ενός δισδιάστατου πίνακα:

```
   double   a[ 5 ][ 7 ];
```

Οι τιμές είναι γραμμένες *κατά γραμμές*. Τι θα πει αυτό; Σε κάθε γραμμή του αρχείου υπάρχουν οι (επτά) τιμές μιας γραμμής του πίνακα. Στην 1η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 0 του πίνακα, στη 2η γραμμή του αρχείου έχουμε τα στοιχεία της γραμμής 1 του πίνακα κ.ο.κ. Πώς διαβάζουμε τις τιμές των στοιχείων του πίνακα;

```
   for ( int r(0); r < 5; ++r )
   {
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][c];
      } // for ( int c . . .
   } // for ( int r . . .
```

Όπως καταλαβαίνεις για κάθε τιμή της *r* διαβάζουμε τις τιμές των στοιχείων της γραμμής *r* του πίνακα. Αυτό γίνεται με την:

```
      for ( int c(0); c < 7; ++c )
      {
         ins >> a[r][s];
      } // for ( int s . . .
```

και φυσικά εδώ *δεν μπορούμε να αλλάξουμε τη σειρά* (φωλιάσματος) των δύο **for**.

Βέβαια, υπάρχει και η περίπτωση να δοθούν οι τιμές των στοιχείων *κατά στήλες*. Δηλαδή, σε κάθε γραμμή του αρχείου υπάρχουν οι (πέντε) τιμές μιας στήλης του πίνακα Στην περίπτωση αυτή είμαστε υποχρεωμένοι να διαβάσουμε ως εξής:

```
   for ( int c(0); c < 7; ++c )
   {
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
   } // for ( int c . . .
```

Τώρα, η

```
      for ( int r(0); r < 5; ++r )
      {
         cin >> a[r][c];
      } // for ( int r . . .
```

διαβάζει τη στήλη *c* του πίνακα.

Παρομοίως γίνεται και το γράψιμο των στοιχείων δισδιάστατου πίνακα κατά γραμμές ή κατά στήλες (άσκ. 8-4).

Παράδειγμα 3 - Συμμετρικότητα τετραγωνικού πίνακα

Ένας τετραγωνικός πίνακας –δηλαδή: δισδιάστατος πίνακας με ίσους αριθμούς γραμμών και στηλών– λέγεται *συμμετρικός* (ως προς την πρώτη διαγώνιο) αν για όλα τα στοιχεία του ισχύει η ιδιότητα $a_{rc} == a_{cr}$. Οι παρακάτω εντολές ελέγχουν αν ο

```
   double a[N][N];
```

(όπου *N* σταθερά τύπου `int` με θετική τιμή) είναι συμμετρικός:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
```

```
   {
      for ( int c(0); c < N; ++c )
      {
         if( a[r][c] != a[c][r] )  symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Πράγματι, κάνουμε την υπόθεση ότι ο a είναι συμμετρικός (`symmet = true`) και στη συνέχεια αν βρούμε κάποιο ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$ διορθώνουμε την τιμή της *symmet* σε `false`.

Αυτό το κομμάτι προγράμματος είναι σπάταλο από άποψη χρόνου επεξεργασίας διότι, πριν απ' όλα:

- Ελέγχει κάθε ζευγάρι δυο φορές, π.χ. ενώ θα ελέγξει αν $a_{23} \neq a_{32}$, αργότερα θα ελέγξει και αν $a_{32} \neq a_{23}$, πράγμα άχρηστο.
- Ελέγχει –και μάλιστα δυο φορές– αν κάθε στοιχείο της πρώτης διαγωνίου είναι ίσο με τον εαυτό του.

Αυτές οι ατέλειες διορθώνονται αν αρχίζουμε την εξέταση της κάθε γραμμής ένα στοιχείο μετά τη διαγώνιο. Αν πάρουμε υπόψη μας ότι το στοιχείο της διαγωνίου στη γραμμή *r* είναι το a_{rr}, θα πρέπει να γράψουμε:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Στη χειρότερη περίπτωση –όταν ο πίνακας *είναι* συμμετρικός– η `if( a[r][c] != a[c][r] )...` θα εκτελεσθεί $\frac{1}{2}N(N-1)^2$ φορές, ενώ στον πρώτο αλγόριθμο θα εκτελεσθεί N^2 φορές.

Υπάρχει όμως και άλλη σπατάλη: αν βρούμε *ένα* ζεύγος a_{rc}, a_{cr} για το οποίο $a_{rc} \neq a_{cr}$, δεν έχει νόημα να συνεχίσουμε τις συγκρίσεις· *ξέρουμε ότι ο πίνακας δεν είναι συμμετρικός*.

Στο παρακάτω κομμάτι έχουμε διορθώσει και αυτήν την ατέλεια:

```
   symmet = true;
   for ( int r(0); symmet && r < N; ++r )
   {
      for ( int c(r+1); symmet && c < N; ++c )
      {
         if ( a[r][c] != a[c][r] ) symmet = false;
      } // for ( int c . . .
   } // for ( int r . . .
```

Δηλαδή, έχουμε πάλι μετρούμενες επαναλήψεις με τη διαφορά ότι τώρα βάλαμε και πρόσθετους ελέγχους για το αν ανιχνεύθηκε παραβίαση της συμμετρίας, οπότε σταματούμε αμέσως.

Μπορούμε να αποφύγουμε τη διπλή συνθήκη στις **for** αν ξαναγράψουμε, χρησιμοποιώντας τη `break`:

```
   symmet = true;
   for ( int r(0); r < N; ++r )
   {
      for ( int c(r+1); c < N; ++c )
      {
         if ( a[r][c] != a[c][r] )
         {  symmet = false;   break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
```

Αν εκτελεσθεί η πρώτη **break** θα διακόψει την εκτέλεση της εσωτερικής **for**. Για να διακόψουμε και την εκτέλεση της εξωτερικής χρειάζεται η δεύτερη **break** που εκτελείται όταν βρει τη *symmet* με τιμή **false**. Αυτή η μετατροπή επιταχύνει σημαντικά το πρόγραμμά μας αφού επιταχύνει την εσωτερική **for** με την απλούστευση της συνθήκης συνέχισης.

Πάντως χρησιμοποιώντας την «καταραμένη» εντολή **goto**, μπορούμε να το κάνουμε ακόμη ταχύτερο:

```
      symmet = true;
      for ( int r(0); r < N; ++r )
      {
         for ( int c(r+1); c < N; ++c )
         {
            if ( a[r][c] != a[c][r] )
            {  symmet = false;  goto lb99;  }
         } // for ( int c . . .
      } // for ( int r . . .
lb99: .....
```

Αυτή είναι μια θεμιτή χρήση της **goto** και κανείς δεν μπορεί να ισχυρισθεί ότι οι άλλες γραφές είναι καλύτερες από αυτήν. Πάντως, πρόσεξε και μια λεπτομέρεια: η εντολή με την ετικέτα **lb99** θα πρέπει να βρίσκεται ακριβώς μετά το τέλος των **for** και όχι οπουδήποτε αλλού μέσα στο πρόγραμμά μας.

Το πρώτο κομμάτι προγράμματος, με τις δύο **for**, περνάει από *όλα* τα στοιχεία του πίνακα (όλοι οι δυνατοί συνδυασμοί των *r* και *c*). Το δεύτερο κομμάτι σαρώνει τα στοιχεία του πίνακα που βρίσκονται *πάνω από την κύρια διαγώνιο* του πίνακα. Τα κομμάτια προγράμματος τρίτο, τέταρτο και πέμπτο σαρώνουν τα στοιχεία του πίνακα που βρίσκονται πάνω από την κύρια διαγώνιο του πίνακα στη χειρότερη περίπτωση.

Παράδειγμα 4 - Πολλαπλασιασμός πινάκων

Έστω ότι ο δισδιάστατος πίνακας a αποτελείται από *l* γραμμές και *m* στήλες, ενώ ο επίσης δισδιάστατος πίνακας b αποτελείται από *m* γραμμές και *n* στήλες. Να υπολογιστεί το γινόμενό τους c, που είναι δισδιάστατος πίνακας *l* γραμμών και *n* στηλών.

Τα στοιχείο του πίνακα c, στη γραμμή *r* και τη στήλη *c* (c_{rc}), υπολογίζεται, όπως γνωρίζουμε από τα μαθηματικά, με τον παρακάτω τύπο:

$$c_{rc} = \sum_{k=0}^{m-1} a_{rk} b_{kc}$$

όπου: $r = 0..l\text{-}1$, $c = 0..n\text{-}1$.

Δηλαδή, το στοιχείο c_{rc} είναι το άθροισμα όλων των όρων $a_{rk}b_{kc}$, όπου ο δείκτης *k* διατρέχει τις τιμές $0..m\text{-}1$.

Αυτός ο υπολογισμός μπορεί να δοθεί στη C++ με μιά **for** και μια εντολή εκχώρησης:

```
      c[r][c] = 0;
      for ( int k(0); k < m; ++k )
         c[r][c] += a[r][k]*b[k][c];
```

Το παρακάτω πρόγραμμα διαβάζει τα στοιχεία των πινάκων a και b, υπολογίζει το γινόμενό τους, δηλ. βρίσκει τα στοιχεία του πίνακα c και μετά τυπώνει τα στοιχεία του γινομένου c (καθώς και των πινάκων a και b), όπως βλέπεις στη συνέχεια:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   const int l = 3,  m = 5,  n = 2;

   int a[ l ][ m ],  b[ m ][ n ];
   int c[ l ][ n ];
```

```
   ifstream atx( "arr.txt" );
// Διάβασε τα στοιχεία του a
   for ( int r(0); r < l; ++r )
      for ( int c(0); c < m; ++c )  atx >> a[r][c];
// Διάβασε τα στοιχεία του b
   for ( int r(0); r < m; ++r )
      for ( int c(0); c < n; ++c )  atx >> b[r][c];
   atx.close();

// Πολλαπλασιασμός Πινάκων
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         c[r][c] = 0;
         for ( int k(0); k < m; ++k )
            c[r][c] += a[r][k]*b[k][c];
      } // for (c . . .
   } // for (r . . .

// Γράψε τα στοιχεία των a, b, c
   cout << " Στοιχεία του πίνακα a" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < m; ++c )
      {
         cout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα b" << endl;
   for ( int r(0); r < m; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << b[r][c] << "  ";
      }
      cout << endl;
   }
   cout << " Στοιχεία του πίνακα c" << endl;
   for ( int r(0); r < l; ++r )
   {
      for ( int c(0); c < n; ++c )
      {
         cout.width(3);  cout << c[r][c] << "  ";
      }
      cout << endl;
   }
}
```

Για να δοκιμάσουμε το πρόγραμμα γράφουμε με τον κειμενογράφο το παρακάτω αρχείο:

```
1    2    3    4    5
6    7    8    9    1
1    4    7    2    5
10   10
12   45
47   18
12   31
18   29
```

και το φυλάγουμε με όνομα "arr.txt" (σε μορφή text). Όταν εκτελεσθεί το πρόγραμμα μας δίνει:

```
 Στοιχεία του πίνακα a
  1    2    3    4    5
  6    7    8    9    1
  1    4    7    2    5
```

```
 Στοιχεία του πίνακα b
 10   10
 12   45
 47   18
 12   31
 18   29
 Στοιχεία του πίνακα c
313  423
646  827
501  523
```

12.5 Η Σειρά Αποθήκευσης

Συνήθως, η σειρά αποθήκευσης των στοιχείων ενός πολυδιάστατου πίνακα θα σου είναι αδιάφορη. Υπάρχει όμως μια τουλάχιστον περίπτωση, που θα δούμε στην επόμενη παράγραφο, όπου σου χρειάζεται. Ας την δούμε λοιπόν.

Όπως αναφέραμε προηγουμένως, «τα στοιχεία ενός δισδιάστατου πίνακα *απόθηκεύονται κατά γραμμές*». Ας δούμε τι σημαίνει αυτό πιο συγκεκριμένα.

Έστω ότι έχουμε δηλώσει:

```
double a[5][7];
```

Η αποθήκευση των στοιχείων ξεκινάει από το a[0][0]. Η επόμενη θέση είναι για το a[0][1], η μεθεπόμενη για a[0][2] κ.ο.κ. Έξη θέσεις μετά το a[0][0] υπάρχει το τελευταίο στοιχείο της γραμμής 0, το a[0][6]. Μετά από αυτό έχουμε τη θέση για το a[1][0], το πρώτο στοιχείο της γραμμής 1. Όπως καταλαβαίνεις, το στοιχείο a[*r*][*c*] βρίσκεται 7*r* + *c* θέσεις μετά το a[0][0].

Προσοχή: ▶
Όταν λέμε «θέση», εννοούμε με την έννοια που είδαμε στην §12.3.2, δηλαδή θέση μνήμης που μπορεί να αποθηκεύσει ένα στοιχείο του πίνακα, στην περίπτωσή μας, μια τιμή τύπου `double` (συχνότατα αυτό σημαίνει 8 ψηφιολέξεις).◀

Ας κάνουμε ένα πείραμα για να επιβεβαιώσουμε τον παραπάνω τύπο. Δίνουμε τιμές στα στοιχεία του a ως εξής:

```
for ( int r(0); r <= 4; ++r )
{
   for ( int c(0); c <= 6; ++c )
   {
      a[r][c] = r + 0.1*c;
   } // for ( int c . . .
} // for ( int r . . .
```

Έτσι, τα στοιχεία της τελευταίας γραμμής (γραμμή 4) παίρνουν τιμές: 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6.

Τώρα, θα προσπαθήσουμε να δούμε τον a ως μονοδιάστατο. Δηλώνουμε:

```
double* p;
```

και βάζουμε:

```
p = &a[0][0];
```

Αφού η C++ καταλαβαίνει τα βέλη και τους πίνακες με τον ίδιο τρόπο, μπορούμε να δούμε το *p* ως μονοδιάστατο πίνακα, με στοιχεία τύπου `double`, που ξεκινάει από το a[0][0]. Από εκεί όμως ξεκινάει η αποθήκευση του a! Με το *p* λοιπόν μπορούμε να χειριστούμε τον a ως μονοδιάστατο πίνακα.

Σύμφωνα με τον τύπο που δώσαμε παραπάνω, τα στοιχεία της γραμμής 4 του a βρίσκονται στις θέσεις από 7×4 + 0 μέχρι 7×4 + 6 μετά το a[0][0]. Ζητούμε λοιπόν και μεις να δούμε τις τιμές αυτών των στοιχείων του πίνακα *p*:

```
r = 4;
for ( int c(0); c <= 6; ++c )
{
```

```
      cout << p[r*7+c] << "  ";
   }
   cout << endl;
```

Αποτέλεσμα; Αυτό ακριβώς που περιμένουμε:

```
4  4.1  4.2  4.3  4.4  4.5  4.6
```

Αν λοιπόν έχεις δηλώσει:

```
   T a[Nr][Nc];
```

τότε:

- ***το στοιχείο* a[*r*][*c*] *βρίσκεται* $r \cdot N_c + c$ *θέσεις (τύπου T) μετά το* a[0][0].**

12.5.1 Τρισδιάστατοι και Πολυδιάστατοι Πίνακες

Γενικότερα, αν έχεις δηλώσει:

```
   T a[N1][N2]...[Nm];
```

τότε:

- ***το στοιχείο* a[k_1][k_2]...[k_m] *βρίσκεται***

  (...($k_1 \times N_2 + k_2$)$\times N_3$ + ... + k_{m-1})$\times N_m + k_m$

  ***θέσεις (τύπου T) μετά το* a[0][0]...[0].**

Δες το παρακάτω πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   int  a[2][2][2];
   int* ip;

   for ( int l(0); l < 2; ++l )
      for ( int r(0); r < 2; ++r )
         for ( int c(0); c < 2; ++c )
            a[l][r][c] = 100*(l+1) + 10*(r+1) + c+1;
   ip = &a[0][0][0];
   for ( int k(0); k < 8; ++k )
      cout << ip[k] << "  ";
   cout << endl;
}
```

Αποτέλεσμα:

```
111  112  121  122  211  212  221  222
```

Στα πρώτα τέσσερα στοιχεία το πρώτο ψηφίο, που βγαίνει από την τιμή της l(+1), έχει τιμή 1. Αυτά είναι τα στοιχεία που βρίσκονται στο «επίπεδο» $l = 0$ ενώ τα τέσσερα τελευταία βρίσκονται στο επίπεδο $l = 1$. Τα δύο πρώτα στοιχεία βρίσκονται στη γραμμή $r = 0$ του επιπέδου $l = 0$.

Το στοιχείο `a[1][0][1]`, που έχει τιμή 212, το βλέπουμε ως ip[((1×2 + 0)×2 + 1], δηλαδή ip[5].

Και το `a[1][0]` τι είναι; Τι είναι στην περίπτωση αυτήν το `a[1]`; Αντί για άλλη απάντηση δοκίμασε τις εντολές:

```
   for ( int l(0); l < 2; ++l )
   {
      cout << a[l] << "  " << &a[l][0][0] << endl;
      for ( int r(0); r < 2; ++r )
         cout  << "   " << a[l][r] << "  " << &a[l][r][0] << endl;
   }
```

που θα δώσουν:

```
0x22ff50  0x22ff50
   0x22ff50  0x22ff50
```

```
   0x22ff58  0x22ff58
0x22ff60  0x22ff60
   0x22ff60  0x22ff60
   0x22ff68  0x22ff68
```

Δηλαδή: το `a[l]` είναι βέλος προς το στοιχείο `a[l][0][0]` ενώ το `a[l][r]` είναι βέλος προς το `a[l][r][0]`.

12.6 Παράμετρος Πίνακας (ξανά)

Ας πούμε ότι έχουμε ορίσει έναν τύπο:

```
typedef char   ProSthlh[13];
```

με στόχο να παραστήσουμε σε στοιχεία αυτού του τύπου στήλες ΠροΠο. Αν δηλώσουμε π.χ.:

```
   ProSthlh c;
```

μπορούμε να δώσουμε:

```
   c[0] = '1';  c[1] = '1';  c[2] = 'X';...
   c[11] = 'X';  c[12] = '2';
```

Ύστερα από αυτό, μπορούμε να παραστήσουμε ένα δελτίο ΠροΠο, με *N* στήλες, με έναν πίνακα:

```
   ProSthlh deltio[N];
```

Θέλουμε μια συνάρτηση που θα παίρνει ένα δελτίο ΠροΠο και τη νικήτρια στήλη και θα μας επιστρέφει, ως τιμή, το πλήθος των στηλών του δελτίου που κερδίζουν (συμφωνούν με τη νικήτρια σε 11 σημεία τουλάχιστον).

Μπορούμε να γράψουμε το εξής σχέδιο:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
      μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      if ( ne >= 11 )  ++es;
   } // for
   return es;
} // winners
```

Η μέτρηση του πλήθους *ne* σωστών σημείων της στήλης deltio[*k*] γίνεται ως εξής:

```
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
```

Και να ολόκληρη η συνάρτηση:

```
int winners( ProSthlh deltio[], int N, ProSthlh nik )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
      {
         if ( deltio[k][j] == nik[j] )  ++ne;
      } // for ( int j . . .
      if ( ne >= 11 )  ++es;
   } // for ( int k . . .
   return es;
```

```
} // winners
```

Πώς την καλούμε; Έστω ότι έχουμε δηλώσει στη **main**:

```
   ProSthlh c, d[138];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners(d, 138, c) << endl;
```

Παρά την προσπάθεια που κάναμε (;) για να κρύψουμε την αλήθεια, αυτή δεν κρύβεται: στη μέτρηση των σωστών σημείων βλέπουμε το δελτίο ως *δισδιάστατο πίνακα*. Να αφήσουμε κατά μέρος τον τύπο ProSthlh και να πούμε τα πράγματα με το όνομά τους;

```
int winners2( char deltio[][13], int N, char nik[] )
{
   int es, ne;

   es = 0;           // πλήθος στηλών με 11 επιτυχίες τουλάχιστον
   for ( int k(0); k < N; ++k )
   {
   // μέτρα το πλήθος ne σωστών σημείων της στήλης deltio[k]
      ne = 0;
      for ( int j(0); j <= 12; ++j )
         if ( deltio[k][j] == nik[j] )  ++ne;
      if (ne >= 11)  ++es;
   } // for ( int k...
   return es;
} // winners2
```

Εδώ βλέπεις πώς περνούμε μια παράμετρο-δισδιάστατο πίνακα: το πλήθος των στηλών (13) γράφεται, το πλήθος των γραμμών όχι απαραίτητα. Αν θέλεις να περάσεις έναν πολυδιάστατο πίνακα θα πρέπει να καθορίζεις όλες τις διαστάσεις εκτός από την πρώτη.

Αν έχουμε δηλώσει στη **main**:

```
   char c[13], d[138][13];
```

Θα πάρουμε το πλήθος στηλών με επιτυχίες ως εξής:

```
   cout << winners2(d, 138, c) << endl;
```

Μέχρι εδώ καλά. Δες τώρα ένα άλλο πρόβλημα.

Ας πούμε τώρα ότι θέλουμε να περάσουμε έναν δισδιάστατο πίνακα χωρίς να έχουμε καθορισμένο πλήθος στηλών· θέλουμε να περάσουμε τα πλήθη γραμμών και στηλών ως παραμέτρους. Πώς μπορεί να γίνει αυτό; Με βάση αυτά που είπαμε στην προηγούμενη παράγραφο. Δηλαδή *περνούμε τον πίνακα ως μονοδιάστατο*. Ας το δούμε με ένα παράδειγμα.

Παράδειγμα

Θέλουμε μια συνάρτηση που θα παίρνει έναν τετραγωνικό πίνακα a με στοιχεία τύπου `double` και θα μας επιστρέφει τιμή `true` αν ο a είναι συμμετρικός και `false` αν δεν είναι.

Η ιδέα είναι η εξής: Όταν καλούμε τη συνάρτηση θα της δίνουμε τη διεύθυνση του πρώτου στοιχείου του πίνακα και πλήθος *N* γραμμών και στηλών. Μέσα στη συνάρτηση δεν θα γράφουμε `a[r][c]` αλλά `a[r*n+c]`.

```
bool isSymmetric( const double* a, int n ) // double a[n][n]
{
   bool symmet( true );

   for ( int r(0); r < n; ++r )
   {
      for ( int c(r+1); c < n; ++c )
      {
//       if ( a[r][c] != a[c][r] ) {  symmet = false;  break;  }
         if ( a[r*n+c] != a[c*n+r] ) {  symmet = false;  break;  }
      } // for ( int c . . .
      if ( !symmet )  break;
   } // for ( int r . . .
   return symmet;
} // isSymmetric
```

Ας δούμε τώρα πώς την καλούμε. Έστω ότι έχουμε:

```
double p1[2][2] = { {1,2}, {3,4} },
       p2[3][3] = { {1, 0, 1.5}, {0, 1.8, 2}, {1.5, 2, 4.1} };
```

Αν θέλουμε να ελέγξουμε τη συμμετρικότητα των p1 και p2 με τη συνάρτησή μας γράφουμε:

```
if ( isSymmetric(&p1[0][0], 2) )  cout << " ο p1 ναι" << endl;
                            else  cout << " ο p1 όχι" << endl;
if ( isSymmetric(&p2[0][0], 3) )  cout << " ο p2 ναι" << endl;
                            else  cout << " ο p2 όχι" << endl;
```

€€€

Όπως βλέπεις –και στην περίπτωση αυτήν– χρησιμοποιούμε αυτά που ξέρουμε για την εσωτερική παράσταση για να περάσουμε ως παράμετρο και να χειριστούμε έναν δισδιάστατο πίνακα. Αυτό είναι μάλλον ένα τέχνασμα αλλά είναι ο μόνος τρόπος που έχουμε.[4]

12.6.1 Και Άλλα Τεχνάσματα

Στην Άσκ. 9-6 ζητούμε μια «*συνάρτηση rawSum() που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο r1 και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής r1.*»

Όποιος έχει διαβάσει με προσοχή αυτά που είδαμε μέχρι τώρα θα πεί: «Δεν χρειάζεται, έχουμε τη *vectorSum*!» Πράγματι, αν

```
double a[5][7], z;
```

και θέλουμε το άθροισμα των στοιχείων της γραμμής 2 μπορούμε να γράψουμε:

```
z = vectorSum( a[2], 7, 0, 6 );
```

ή

```
z = vectorSum( &a[2][0], 7, 0, 6 );
```

Εδώ στηριζόμαστε στο πώς αποθηκεύει η C++/C τους δισδιάστατους πίνακες και θεωρείται τέχνασμα. Λύσε λοιπόν την Άσκ. 9-6 με πιο «ορθόδοξο» τρόπο.

Ας δούμε όμως και μια περίπτωση που τα τεχνάσματα δεν δουλεύουν. Έχουμε τον πίνακα:

```
double p3[6][9] = { {1, 2, 4, 0, 5, 2, 7, 9, 0},
                    {2, 2, 0, 3, 4, 5, 9, 7, 8},
                    {4, 0, 2, 6, 3, 4, 0, 5, 7},
                    {3, 4, 8, 3, 5, 7, 0, 4, 1},
                    {4, 5, 1, 4, 7, 9, 0, 5, 7},
                    {1, 1, 5, 0, 0, 0, 8, 8, 6} };
```

και θέλουμε να ελέγξουμε αν οι (τετραγωγικοί) υποπίνακες από `p3[0][0]` μέχρι `p3[2][2]` και από `p3[2][3]` μέχρι `p3[5][6]` είναι συμμετρικοί. Θα μπορούσαμε να ρωτήσουμε *isSymmetric*`(&p3[0][0], 3)` και *isSymmetric*`(&p3[2][3], 4)`; Ούτε για αστείο!

Αν μας ενδιαφέρουν τέτοιες επεξεργασίες ξαναγράφουμε την *isSymmetric* ώστε να τηρούμε αυτό που είπαμε και πιο πριν:

- ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Στο παράδειγμά μας η *isSymmetric* θα πρέπει να ξαναγραφεί ως εξής:

```
bool isPartSymmetric( const double* a, int nRow, int nCol,
                      int rUL, int cUL, int rLR, int cLR )
```

Οι παράμετροι της πρώτης γραμμής περνούν στη συνάρτηση τον πίνακα: αρχή, γραμμές, στήλες. Οι παράμετροι της δεύτερης γραμμής περνούν την περιοχή επεξεργασίας που τη θεωρούμε ως ορθογώνιο και περνούμε: τη γραμμή (*rUL*) και τη στήλη (*cUL*) πάνω αριστε-

[4] Αργότερα θα μάθουμε τους δυναμικούς πίνακες που έχουν πιο «ορθόδοξο» χειρισμό.

ρής κορυφής και τη γραμμή (*rLR*) και τη στήλη (*cLR*) κάτω δεξιάς κορυφής. Για τα παραδείγματα που είπαμε πιο πάνω θα καλούμε ως εξής:

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          0, 0, 2, 2 ) . . .
```

και

```
   . . . isPartSymmetric( &p3[0][0], 6, 9,
                          2, 3, 5, 6 ) . . .
```

12.7 Οι Παράμετροι της `main`

Ας πούμε ότι έχουμε ένα πρόγραμμα φυλαγμένο στο αρχείο mainargs.exe. Για να ζητησουμε την εκτέλεσή του από γραμμή εντολών δίνουμε:

```
D:\>mainargs<enter>
```

Όπως θα έχεις δει όμως, μερικές φορές εκτός από το όνομα του αρχείου γράφουμε και διάφορες παραμέτρους, οι οποίες περνούν στο πρόγραμμα και επηρεάζουν την εκτέλεσή του. Μπορούμε να γράψουμε τέτοια προγράμματα; Ναι! Ας πούμε ότι το αρχικό πρόγραμμα αυτού που έχουμε στο mainargs.exe είναι το παρακάτω:

```
#include <iostream>
using namespace std;
int main( int argc, char* argv[] )
{
   cout << "Η τιμή της argc είναι " << argc << endl;
   cout << "Αυτά είναι τα ορίσματα που πέρασαν στην int main:"
        << endl;

   for ( int k(0); k <= argc; ++k )
      cout << "   argv[" << k << "]: " << argv[k] << endl;
} // main
```

Η διαφορά του από αυτά που είδαμε μέχρι τώρα βρίσκεται στις παραμέτρους που υπάρχουν στην επικεφαλίδα της **main**. Η δεύτερη παράμετρος είναι ένας πίνακας που κάθε του στοιχείο είναι ένα «C-style string»· μπορεί να τη δεις και ως `char** argv`. Η πρώτη παράμετρος μας λέει πόσα στοιχεία έχει πίνακας (+1)· για την ακρίβεια μας λέει τον δείκτη του τελευταίου στοιχείου.

Τι μπορεί να είναι αυτό το "`char* argv[]`"; Όπως είναι γραμμένο, μας λέει ότι έχουμε πίνακα που το κάθε στοιχείο του είναι τύπου "`char*`". Δηλαδή, μπορεί να είναι δισδιάστατος πίνακας με στοιχεία τύπου `char`; Ναι, αλλά κάπως διαφορετικός από αυτούς που είδαμε. Παρ' όλα αυτά μπορούμε να τον αξιοποιήσουμε με αυτά που μάθαμε.

Όπως βλέπεις, αυτό που κάνει το πρόγραμμά μας είναι να τυπώνει τις τιμές των παραμέτρων. Να ένα παράδειγμα εκτέλεσης: Ζητούμε να εκτελεστεί ως εξής:

```
E:\cpp2bk\progs>mainargs όρισμα1 a\b "όρισμα 3" 4 5.5 c\d?'ef<enter>
```

και παίρνουμε:

```
Η τιμή της argc είναι 7
Αυτά είναι τα ορίσματα που πέρασαν στην int main:
   argv[0]: mainargs
   argv[1]: όρισμα1
   argv[2]: a\b
   argv[3]: όρισμα 3
   argv[4]: 4
   argv[5]: 5.5
   argv[6]: c\d?'ef
   argv[7]:
```

Ας δούμε λοιπόν τις τιμές των στοιχείων του argv:

- Το `argv[0]` είναι το όνομα του αρχείου που περιέχει το (εκτελέσιμο) πρόγραμμα (μπορεί να δεις και την πλήρη διαδρομή (path) προς αυτό).

- Το `argv[1]` δείχνει τον πρώτο ορμαθό χαρακτήρων μετά το όνομα του αρχείου, δηλαδή το πρώτο όρισμα (στην περίπτωσή μας: `όρισμα1`).
- Το `argv[`*k*`]` δείχνει τον *k*-στό ορμαθό χαρακτήρων μετά το όνομα του αρχείου.
- Το `argv[argc-1]` δείχνει το τελευταίο όρισμα (ορμαθό χαρακτήρων).
- Το `argv[argc]` είναι φρουρός: περιέχει το (βέλος) 0.

Όπως βλέπεις, τα ορίσματα-ορμαθοί ξεχωρίζουν μεταξύ τους με ένα διάστημα. Στο τρίτο όρισμα θέλαμε να περάσουμε την τιμή `όρισμα 3`, που περιέχει διάστημα. Γι' αυτό υποχρεωθήκαμε να το γράψουμε: `"όρισμα 3"`.

12.8 Τελικώς ...

Τα βέλη είναι εργαλεία με τα οποία μπορείς να χειρίζεσαι πίνακες. Η χρήση βελών (υποτίθεται ότι) κάνει τα προγράμματά σου πιο γρήγορα αλλά η απλή κωδικοποίησή τους με δείκτες τα κάνει πιο ευανάγνωστα (και πιο σίγουρα). Προτίμησε λοιπόν τους δείκτες και χρησιμοποίησε βέλη μόνον όταν δεν μπορείς να τα αποφύγεις.

Δεν μπορείς να τα αποφύγεις (προς το παρόν) όταν περνάς πολυδιάστατο πίνακα σε συνάρτηση.

Όταν περνάς πίνακα σε συνάρτηση μην προσπαθείς να ξεγελάσεις τον μεταγλωττιστή με τεχνάσματα. Επαναλαμβάνουμε για τρίτη φορά τη συμβουλή που δώσαμε:

♦ ***Όταν γράφουμε συνάρτηση που μπορεί να επεξεργάζεται τμήμα πίνακα περνούμε σε αυτήν μέσω παραμέτρων α) τον πίνακα και β) το τμήμα του που μας ενδιαφέρει.***

Ασκήσεις

Α Ομάδα

12-1 Τι δίνει το παρακάτω πρόγραμμα; (εκτέλεσέ το *εσύ, όχι ο υπολογιστής*)

```
#include <iostream>
using namespace std;

int qaw( int* p )
{
   *p = 5;
   return *p;
} // qaw

int main()
{
   int x( 0 );
   cout << " the value of x is " << x << endl;
   int y( qaw(&x) );
   cout << " the new value of x is " << x << endl;
   cout << " the value of y is " << y << endl;
} // main
```

12-2 Τα ίδια για το πρόγραμμα:

```
#include <iostream>
using namespace std;
int main()
{
   char a[] = "abcdefgh";
   for ( char* p(a); *p != 0; ++p )  cout << p << endl;
} // main
```

12-3 Έστω ότι έχουμε έναν τετραγωνικό πίνακα πραγματικών αριθμών. Το άθροισμα των στοιχείων της (κύριας) διαγωνίου ονομάζεται **ίχνος** (trace) του πίνακα. Γράψε συνάρτηση *trace* που θα τροφοδοτείται με τον πίνακα και θα υπολογίζει και θα επιστρέφει το ίχνος.

12-4 Γράψε συνάρτηση *trace2* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της *δεύτερης* διαγωνίου.

12-5 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων δύο στηλών *c1, c2* ενός δισδιάστατου πίνακα. Το ίδιο για τις τιμές των στοιχείων δύο γραμμών *r1, r2*.

12-6 Γράψε συνάρτηση *rawSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και έναν φυσικο *r1* και θα υπολογίζει και θα επιστρέφει το άθροισμα των στοιχείων της γραμμής *r1*.

Γράψε συνάρτηση *colSum()* που θα κάνει το ίδιο για τα στοιχεία της στήλης *c1*.

12-7 Γράψε συνάρτηση *isDiagonal()* που θα τροφοδοτείται με τετραγωνικό πίνακα πραγματικών αριθμών και θα επιστρέφει τιμή `true` αν και μόνον αν ο πίνακας είναι διαγώνιος (όλα τα στοιχεία εκτος της *πρώτης* διαγωνίου είναι μηδέν).

Β Ομάδα

12-8 Τροποποίησε το πρόγραμμα της Άσκ. 9-2 ώστε να δίνει:

```
h
gh
fgh
efgh
defgh
cdefgh
bcdefgh
abcdefgh
```

Υπόδ.: Αν δεν μπορείς να το κάνεις με βέλη, κάνε το πρώτα όπως μπορείς και μετά μετάτρεψέ το ώστε να χρησιμοποιεί μόνο βέλη.

12-9 Γράψε συνάρτηση *periphSum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών με *nR* γραμμές και *nC* στήλες και θα υπολογίζει και θα επιστρέφει το άθροισμα των περιφερειακών στοιχείων (των γραμμών 0 και *nR*-1 και των στηλών 0 και *nC*-1.)

12-10 Γράψε συνάρτηση *matrix2Sum()* που θα τροφοδοτείται με δισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε συνάρτηση *matrix3Sum()* που θα τροφοδοτείται με τρισδιάστατο πίνακα πραγματικών αριθμών και θα υπολογίζει και θα επιστρέφει το άθροισμα όλων των στοιχείων του.

Γράψε πρόγραμμα που θα δείχνει τον τρόπο χρήσης των δύο συναρτήσεων.

12-11 Γράψε πρόγραμμα που θα διαβάζει από το πληκτρολόγιο τις (πραγματικές) τιμές των στοιχείων ενός δισδιάστατου πίνακα και θα υπολογίζει και θα μας δίνει:

- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *γραμμών* του,
- το μέγιστο από τα αθροίσματα των απολύτων τιμών των στοιχείων των *στηλών* του.

Γ Ομάδα

12-12 Γράψε εντολές που θα αντιμεταθέτουν τις τιμές των στοιχείων της *k* γραμμής με αυτά της *k* στήλης ενός τετραγωνικού πίνακα. Δοκίμασε να κάνεις το ίδιο για τα στοιχεία της *r* γραμμής και της *c* στήλης όταν $r \neq c$.

12-13 Γράψε την *isPartSymmetric()* που προδιαγράψαμε στην§12.6.1.