κεφάλαιο

16

Δυναμική Παραχώρηση Μνήμης

Ο στόχος μας σε αυτό το κεφάλαιο:
Να γνωρίσεις το σύστημα διαχείρισης δυναμικής μνήμης της C++. Το σύστημα αυτό είναι εξαιρετικώς ευέλικτο και δίνει τη δυνατότητα στον προγραμματιστή να χρησιμοποιεί τη δυναμική μνήμη –με πίνακες ή δυναμικές δομές δεδομένων– με πολύ αποδοτικό τρόπο. Το βέλος είναι το βασικό εργαλείο.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να χρησιμοποιείς δυναμικούς πίνακες και δυναμικές δομές δεδομένων στα προγράμματά σου. Θα μπορείς να χρησιμοποιείς και βέλη (αλλά και να τα αποφεύγεις όποτε δεν είναι απαραίτητα).

Έννοιες κλειδιά:

- *τελεστές* "`new`", "`new[]`"
- *τελεστές* "`delete`", "`delete[]`"
- *δυναμική μεταβλητή*
- *δυναμικός πίνακας*
- *μεταβλητή-βέλος (pointer)*
- *RAII*

Περιεχόμενα:

16.1 Οι Τελεστές "new" και "delete" ... 494
16.2 Συντακτικά και Βασικές Έννοιες ... 496
16.3 Τιμές Βελών και Δυναμικών Μεταβλητών ... 498
16.4 Δυναμικοί Πίνακες ... 500
16.5 * Η Τρίτη Μορφή του "new" ... 504
16.6 Η Εξαίρεση *bad_alloc* ... 504
16.6.1 Μια Εξήγηση για τις Εξαιρέσεις μας ... 505
16.7 Τα Προβλήματα της Δυναμικής Μνήμης ... 505
16.7.1 RAII: Μια Καλύτερη Λύση ... 509
16.8 Προβλήματα και στις Δομές ... 511
16.9 Δισδιάστατοι Δυναμικοί Πίνακες ... 511
16.10 * Τύπος Βέλους: "void*" ... 515
16.11 * Αναμνήσεις από τη C: *malloc()*, *free()*, *realloc()* ... 517
16.12 Για να Μη Ζηλεύουμε τη *realloc()* ... 519
16.13 Παραδείγματα ... 520
16.13.1 Το Περίγραμμα *linSearch()* ... 532
16.13.2 Χωρίς τη *linSearch()* ... 533
16.13.3 "*reserved* + *incr*" ή "2**reserved*" ... 535
16.14 Προβλήματα Ασφάλειας ... 535
16.15 Ανακεφαλαίωση ... 537
Ασκήσεις ... 538

Α Ομάδα..538

Εισαγωγικές Παρατηρήσεις:

Όπως λέγαμε στην §11.1 «*σε κάθε πρόγραμμα παραχωρούνται τρεις περιοχές μνήμης:*

- *η **στατική**, όπου υλοποιούνται οι καθολικές μεταβλητές (και μερικές άλλες που θα δούμε στη συνέχεια),*
- *η **αυτόματη** (automatic) ή μνήμη **στοίβας** (stack), όπου υλοποιούνται οι τοπικές μεταβλητές των σύνθετων εντολών και των συναρτήσεων και*
- *η **δυναμική** (dynamic) μνήμη που θα δούμε στη συνέχεια.*»

Η ποσότητα *στατικής μνήμης* που θα χρησιμοποιηθεί καθορίζεται όταν γράφουμε το πρόγραμμά μας, όταν δηλώνουμε τις καθολικές και τις στατικές μεταβλητές που χρησιμοποιεί.

Η ποσότητα *μνήμης στοίβας* που θα χρησιμοποιηθεί εξαρτάται

- από τις μεταβλητές που δηλώνουμε στις διάφορες συναρτήσεις και
- από τον τρόπο που θα κληθούν οι συναρτήσεις και πόσες φορές.

Αν, ας πούμε, έχουμε δύο συναρτήσεις *f()* και *g()* και τις καλέσουμε από τη `main` με τις εντολές:

```
f( . . . );
. . .
g( . . . );
```

τότε η μέγιστη ποσότητα αυτόματης μνήμης που θα απαιτηθεί είναι αυτή της `main` και η μέγιστη από τις απαιτούμενες για τις *f()* και *g()*. Αν όμως η `main` καλεί την *f()* και αυτή καλεί τη *g()*, τότε θα χρειαστούμε μνήμη για τη `main` και μνήμη για την *f()* και μνήμη για τη *g()* ταυτοχρόνως.

Αν η *f()* είναι αναδρομική και καλέσει *n* φορές τον εαυτόν της τότε θα χρειαστούμε μνήμη για τη `main` και *n*+1 φορές μνήμη για την *f()*.

Η C++, όπως και άλλες γλώσσες προγραμματισμού, μας επιτρέπει «να παίρνουμε» πρόσθετη μνήμη –τη **δυναμική** μνήμη– σύμφωνα με ανάγκες που παρουσιάζονται *όταν το πρόγραμμά μας εκτελείται*. Επιτρέπει δηλαδή **δυναμική παραχώρηση μνήμης** (dynamic memory allocation).

Το πρότυπο της C++ αναφέρεται στη δυναμική μνήμη με δύο διαφορετικά ονόματα: Αποκαλεί

- **μνήμη σωρού** (heap memory) αυτήν που χειριζόμαστε με τις συναρτήσεις της C (*malloc*, *calloc*, *realloc*, *free*) και
- **free store** αυτήν που χειριζόμαστε με τους τελεστές `new` και `delete` της C++.

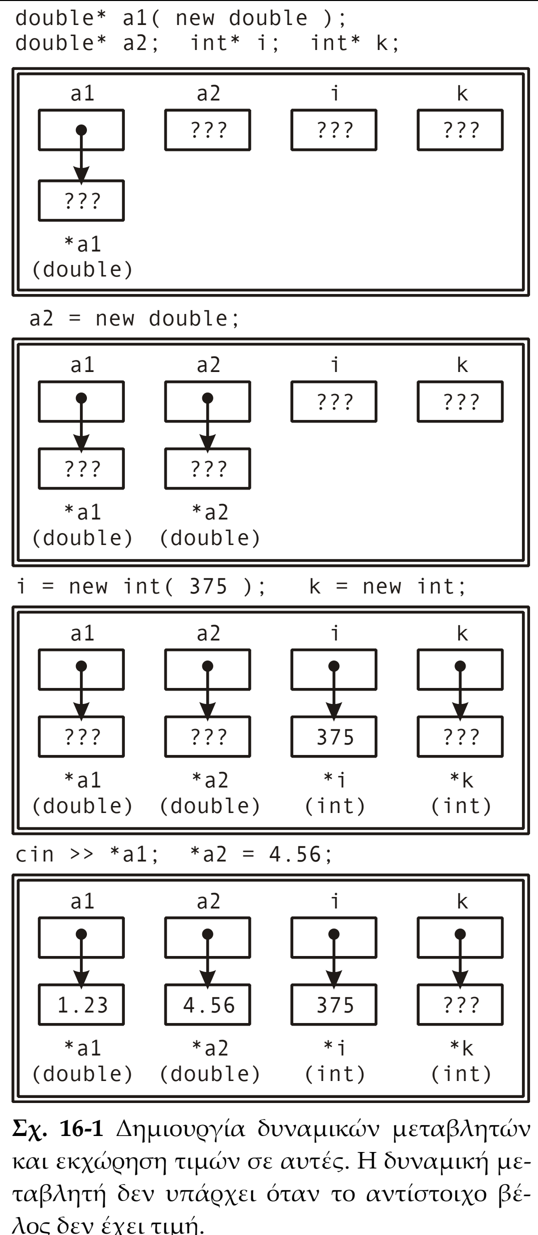
Έτσι, αν κάνεις ένα πείραμα σαν αυτό της §14.8, με αυτά που θα δούμε στο κεφάλαιο αυτό, μπορεί να δεις διαφορετικές περιοχές διευθύνσεων. Αυτό δεν έχει και μεγάλη σημασία αρκεί να τηρείς τον πολύ βασικό κανόνα, που θα επαναλάβουμε και στη συνέχεια: *Μνήμη που παίρνεις με* `new` *θα απελευθερώνεται με* `delete` *και μνήμη που παίρνεις με malloc, calloc, realloc θα απελευθερώνεται με free.*

Θα πρέπει πάντως να επισημάνουμε μια διαφορά της C++ από τις «αδελφές» της απογόνους της C, τη Java και τη C#: Η C++ κράτησε τον τρόπο χειρισμού της δυναμικής μνήμης με *μεταβλητές-βέλη* (pointers) αν και έχει (και) νέα εργαλεία πέρα από αυτά της C.

16.1 Οι Τελεστές “`new`” και “`delete`”

Η δυναμική παραχώρηση μνήμης γίνεται με τους τελεστές “`new`” και “`delete`” και με δύο κατηγορίες μεταβλητών:

- τις **μεταβλητές-βέλη** (pointer variables) –που ήδη ξέρουμε– και



Σχ. 16-1 Δημιουργία δυναμικών μεταβλητών και εκχώρηση τιμών σε αυτές. Η δυναμική μεταβλητή δεν υπάρχει όταν το αντίστοιχο βέλος δεν έχει τιμή.

- τις **δυναμικές** (dynamic) **μεταβλητές**.

Στις τιμές-βέλη αναφερθήκαμε για πρώτη φορά στο Κεφ. 2 (§2.8.3) λέγαμε ότι: «*γράφοντας* `&number` *παίρνουμε τη διεύθυνση μιας θέσης μνήμης όπου υπάρχει η πληροφορία που θέλουμε· παίρνουμε δηλαδή μια παραπομπή προς αυτό που μας ενδιαφέρει. Λέμε λοιπόν ότι η* `&number` *είναι μια* ***παραπέμπουσα*** *ή* ***αναφερόμενη*** *(referencing) τιμή –αφού παραπέμπει ή αναφέρεται σε κάτι– ή τιμή-****βέλος*** *(pointer) – αφού δείχνει κάτι.*» Στην ίδια παράγραφο λέγαμε ακόμη: «*Ας πούμε ότι έχουμε μια τιμή-βέλος p, δηλαδή μια διεύθυνση· πώς μπορούμε να δούμε την τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p; Ή, με άλλα λόγια, ποια είναι αντίστροφη πράξη της "*`&`*"; Η C++ τη συμβολίζει με "*`*`*": η τιμή που είναι αποθηκευμένη στη θέση που μας δείχνει η p παριστάνεται με "*`*p`*". [...] Αν πάρουμε την* `*(&number)` *είναι σαν να παίρνουμε τη* `number`*. Λέμε ότι ο τελεστής "*`*`*"* ***απο-παραμέμπει*** *(dereferences) την τιμή-βέλος στην οποία δρα.*»

Στο Κεφ. 12 είδαμε ότι «*Για τη C++, ένας πίνακας είναι ένα βέλος που δείχνει (έχει ως τιμή) τη θέση της μνήμης όπου αρχίζει η αποθήκευση των στοιχείων του.*» Και (ενώ ένας πίνακας είναι ένα σταθερό βέλος) είδαμε ότι μπορούμε να έχουμε και μεταβλητές-βέλη. Πάντως, σε όλες τις περιπτώσεις οι τιμές και οι μεταβλητές-βέλη έδειχναν θέσεις συμβατικής (στατικής ή αυτόματης) μνήμης.

Τώρα ας δούμε μια άλλη δυνατότητα. Αν έχουμε δηλώσει:

`T* q;` ή (`T *q;`)

αν δηλαδή η *q* είναι μια *μεταβλητή-βέλος*, που δείχνει θέσεις μνήμης τύπου *T*, τότε η εκτέλεση της εντολής

```
q = new T;
```

έχει τα εξής αποτελέσματα:

- Παραχωρείται στο πρόγραμμά μας μια θέση μνήμης τύπου *T*. Αυτή είναι μια *δυναμική μεταβλητή* και μπορούμε να την χρησιμοποιούμε στο πρόγραμμά μας με το «όνομα» `*q`.
- Η *διεύθυνση* της `*q` είναι αποθηκευμένη ως τιμή της *q*.

Αν, αργότερα, δεν χρειαζόμαστε αυτή τη θέση, την «αναλυκλώνουμε» ή την «επιστρέφουμε» με την

```
delete q;
```

Πριν δώσουμε παράδειγματα όπου θα φαίνεται η χρησιμότητα της δυναμικής παραχώρησης μνήμης, θα εισαγάγουμε τα στοιχεία της γλώσσας που απαιτούνται με πιο απλά παραδείγματα.

Πλαίσιο 16.1

Τελεστής new

Αν έχουμε δηλώσει:

```
T*  pv;
```

1) η

```
pv = new T;
```

έχει ως αποτέλεσμα να παραχωρηθεί στο πρόγραμμα μια θέση μνήμης (μεταβλητή) τύπου *T*. Η μεταβλητή είναι η

```
*pv
```

και μπορούμε να τη διαχειριζόμαστε όπως οποιαδήποτε μεταβλητή τύπου *T*. Η διεύθυνση της θέσης που μας παραχωρήθηκε αποθηκεύεται ως τιμή της *pv*.

2) η

```
pv = new T[N];
```

έχει ως αποτέλεσμα να παραχωρηθούν στο πρόγραμμα *N* θέσεις μνήμης τύπου *T*, δηλαδή ένας *πίνακας* με *N* στοιχεία τύπου *T*. Τα στοιχεία του πίνακα είναι:

```
pv[0], pv[1],… pv[N-1]
```

και μπορούμε να τα διαχειριζόμαστε σαν να είχαμε δηλώσει στο πρόγραμμα:

```
T pv[N];
```

16.2 Συντακτικά και Βασικές Έννοιες

Ας πούμε ότι έχουμε κάποιον τύπο *T* και δηλώνουμε τη μεταβλητή:

```
T*  q;
```

Ή, αν θέλεις, ορίζουμε έναν τύπο:

```
typedef  T*   PT;
```

και δηλώνουμε:

```
PT  q;
```

Σε κάθε περίπτωση, η *q* είναι μια *μεταβλητή-βέλος*. Ο *T** ή ο *PT* είναι ένας **τύπος βέλους** με **τύπο στόχο** (target ή domain ή base type) τον *T*.

Όπως ξέρουμε *στην q αποθηκεύονται πληροφορίες που καθορίζουν μια διεύθυνση στην κύρια μνήμη του υπολογιστή*, συνήθως η ίδια η διεύθυνση. Στη διεύθυνση αυτήν, που μας δείχνει η τιμή της *q*, μπορεί να αποθηκευτεί μια τιμή τύπου *T*.

Για παράδειγμα, μετά τα

```
typedef  int*   PInt;
// . . .
   double *a1, *a2;
   PInt   i, j, k;
```

Οι *a*1, *a*2, *i*, *j*, *k* είναι μεταβλητές-βέλη· οι δύο πρώτες προς μεταβλητές τύπου `char`, οι τρεις τελευταίες προς μεταβλητές τύπου `int`.

Παρατήρηση: ▶

Πρόσεξε το εξής: Μετά τον ορισμό «*PInt* είναι ο `int*`», με τη δήλωση που κάνουμε, όλες οι μεταβλητές –*i*, *j*, *k* – είναι τύπου `int*`. Στην πρώτη δήλωση, αν γράφαμε "`double* a1, a2`" η *a*1 θα ήταν τύπου `double*` αλλά η *a*2 θα ήταν τύπου `double`.◀

Θα χρησιμοποιήσουμε αυτές τις (συμβατικές) μεταβλητές για να πάρουμε και να ελέγξουμε δυναμικές μεταβλητές.

Μετά τη δήλωσή τους οι μεταβλητές αυτές είναι αόριστες. Θα τους δώσουμε τιμές με μια παράσταση **new** (Πλ. 16.1). Η πρώτη μορφή της παράατασης είναι γενικώς:

"`new`", *τύπος* ["(", *παράσταση*, ")"];

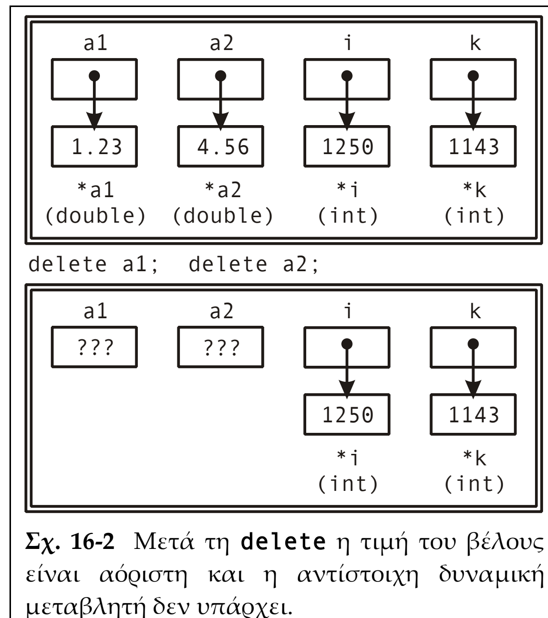
και επιστρέφει ένα βέλος προς μια μεταβλητή οργανωμένη όπως καθορίζει ο «*τύπος*». Αν υπάρχει η «*παράσταση*» θα πρέπει να δίνει αποτέλεσμα που μπορεί να μετατραπεί σε τιμή του τύπου της δυναμικής μεταβλητής. Στην πιο απλή περίπτωση μπορείς να δώσεις π.χ.:

```
k = new int;
```

Η μεταβλητη-βέλος *k* ορίζεται και δείχνει μια θέση μνήμης –μια δυναμική μεταβλητή– τύπου `int` που παραχωρήθηκε στο πρόγραμμά μας. Πώς τη χειριζόμαστε; Μα ως "`*k`"! *Μετά την εκτέλεση της εντολής ορίζεται η k αλλά η *k είναι αόριστη.*

Μπορούμε να δώσουμε και

```
i = new int( 375 );
```



Σχ. 16-2 Μετά τη **`delete`** η τιμή του βέλους είναι αόριστη και η αντίστοιχη δυναμική μεταβλητή δεν υπάρχει.

Τώρα, ορίζεται η μεταβλητη-βέλος *i* και δείχνει μια δυναμική μεταβλητή (**i*) τύπου `int` *αλλά είναι εξ αρχής ορισμένη και η *i* που δημιουργήθηκε με αρχική τιμή "`375`".

Μπορούμε όμως να δώσουμε τη "`new`" και στη δήλωση:

```
double *a1( new double );
```

Τώρα η *a*1 είναι εξ αρχής ορισμένη. Φυσικά, μπορούμε να έχουμε εξ αρχής ορισμένη και τη δυναμική μεταβλητή (**a*1) αν δώσουμε ως αρχική τιμή στο *a*1 "`new double(1.23)`".

Όλα αυτά μπορείς να τα δεις πιο παραστατικά στο Σχ. 16-1. Όπως βλέπεις, ενώ η *k*, ας πούμε, είναι μια *συμβατική μεταβλητή* η **k* είναι πράγματι *δυναμική*. Οι μεταβλητές-βέλη των παραδειγμάτων υπάρχουν από τη στιγμή που αρχίζει να εκτελείται το πρόγραμμά μας (ή η συνάρτηση) όπου έχουν δηλωθεί. Αλλά οι δυναμικές μεταβλητές δεν υπάρχουν μέχρι να παραχωρηθούν από τον τελεστή `new`.

Αν έχουμε τύπο δομής, ας πούμε τον *Date*, μπορούμε να γράψουμε:

```
Date* pd( new Date(Date(2008, 7, 9)) );
```

Δηλαδή: δώσε μου μνήμη για μια δυναμική μεταβλητή τύπου *Date* (που θα τη δείχνει το βέλος *pd*) με αρχική τιμή αυτήν που δημιουργεί ο δημιουργός του τύπου *Date* αν τροφοδοτηθεί με τις τιμές 2008, 7, 9. Πάντως μπορείς να έχεις το ίδιο αποτέλεσμα και με πιο απλό γράψιμο:

```
Date* pd( new Date(2008, 7, 9) );
```

Κατά τα άλλα:

- ***Χειριζόμαστε τις δυναμικές μεταβλητές όπως όλες τις άλλες του ίδιου τύπου.***

Μπορούμε, για παράδειγμα, να τους δίνουμε τιμή ή να αλλάζουμε την τιμή που έχουν με εντολή εκχώρησης

```
*i = (*i)*2 + 500;
```

ή με εντολή εισόδου από αρχείο ή απο την οθόνη:

```
cin >> *k;
```

Για να μιλάμε όμως για δυναμική μνήμη δεν φτάνει μόνον η δυνατότητα παραχώρησης· πρέπει να έχουμε και δυνατότητα επιστροφής όταν δεν τη χρειαζόμαστε. Ο «αντίστροφος» του `new` είναι ο τελεστής **`delete`** (Πλ. 16.2).

Ας πούμε ότι μετά τη χρήση των δυναμικών μεταβλητών που βλέπουμε στο Σχ. 16-1 βάζουμε τις εντολές

```
delete a1;   delete a2;
```

Πλαίσιο 16.2

Τελεστής `delete`

Αν έχουμε δηλώσει:

```
T*  pv;
```

και έχουμε πάρει δυναμική μνήμη με την

```
pv = new T;
```

η

```
delete pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει η δυναμική μεταβλητή **pv* β) η *pv* να γίνει αόριστη.

Αν έχουμε πάρει δυναμική μνήμη με την:

```
pv = new T[N];
```

η

```
delete[] pv;
```

έχει ως αποτέλεσμα α) να πάψει να υπάρχει ο δυναμικός πίνακας που έδειχνε η *pv* β) η *pv* να γίνει αόριστη.

Στο Σχ. 16-2 βλέπεις δυο στιγμιότυπα από τη συνέχεια της εκτέλεσης.

Οι a1 και a2 δεν είναι πια ορισμένες. Η "`delete a1`" «εξαφανίζει» τη θέση μνήμης (μεταβλητή) **a1* και η "`delete a2`" την **a2*. Τί θα πει «εξαφανίζει»; Οι **a1* και **a2* δεν ελέγχονται πια από το πρόγραμμά μας αλλά από το μηχανισμό διαχείρισης της δυναμικής μνήμης. Αυτό σημαίνει ότι η θέση αυτή μπορεί να παραχωρηθεί ξανά με μια επόμενη "`new` …".

Και τώρα προσοχή:

- ***Αμέσως μετά τη "`delete q`" η *q δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.***

Δυστυχώς, η C++ θα αφήσει την τήρηση αυτού του κανόνα στον προγραμματιστή και δεν θα αποπειραθεί να αποτρέψει μια τέτοια παρανομία.

16.3 Τιμές Βελών και Δυναμικών Μεταβλητών

Όπως ήδη ξέρεις, η C++ μας επιτρέπει να τυπώσουμε την τιμή ενός βέλους ή οποιαδήποτε διεύθυνση. Ας πάμε λοιπόν σε αυτά που είδαμε στο Σχ. 16-1 για να δούμε τι θα μπορούσαν να είναι εκείνα τα "???":

```
#include <iostream>
using namespace std;
int main()
{
   double* a1( new double );
   double* a2;  int* i;  int* k;
   cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
```

Αποτέλεσμα; Κάτι σαν

```
0x3e2478  0x34  0x22ffa8  0x7c800000
```

Αν προσπαθήσεις να κάνεις αποπαραπομπές, για παράδειγμα με την:

```
   cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

είναι πολύ πιθανό να έχεις διακοπή της εκτέλεσης του προγράμματός σου.

Οι αποπαραπομπές γίνονται με ασφάλεια μόνον όταν έχεις ορίσει όλες τις μεταβλητές-βέλη:

```
   a2 = new double;
   i = new int( 375 );   k = new int;
```

```
cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

Αποτέλεσμα κάτι σαν

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
0  1.66976e-307  375  4064328
```

Συγκρίνοντας τα τελευταία αποτελέσματα με τα αρχικά βλέπουμε τα εξής: Μετά τις “`new`”

- Οι διευθύνσεις που είναι αποθηκευμένες στα τέσσερα βέλη είναι «κοντινές», στην ίδια περιοχή της μνήμης.
- Οι αποπαραπομπές γίνονται με ασφάλεια αλλά, φυσικά, η μόνη τιμή που έχει νόημα είναι αυτή του **i* (375).

Την πρώτη φορά, η μόνη διεύθυνση που είχε νόημα ήταν η τιμή της *a*1 που ήταν ορισμένη από τη δήλωση. Στις άλλες μεταβλητές-βέλη τι είχαμε; *Το τυχαίο περιεχόμενό τους ερμηνευμένο ως διεύθυνση*. Έτσι, όταν ζητούσαμε να γίνει μια αποπαραπομπή αν η υποτιθέμενη διεύθυνση ήταν έξω από τον *χώρο διευθύνσεων* του προγράμματος είχαμε διακοπή της εκτέλεσής του.

Σημείωση: ▶

Χώρος διευθύνσεων (address space) ενός προγράμματος είναι το σύνολο των διευθύνσεων που μπορεί «νομίμως» να χρησιμοποιήσει όταν εκτελείται. Στα σύγχρονα ΛΣ, που χρησιμοποιούν τεχνολογίες **εικονικής** ή **υπερβατικής** (virtual) μνήμης αυτές οι διευθύνσεις δεν είναι φυσικές αλλά εικονικές που απεικονίζονται στις φυσικές με κάποιον τρόπο[1]. Αυτό εξηγεί και το ότι σε διαφορετικές εκτελέσεις του ίδιου προγράμματος μπορεί να βλέπεις μερικές (εικονικές) διευθύνσεις –όπως εδώ η τιμής της *a*1– να παραμένουν ίδιες.

Δηλαδή, ολη η μνήμη που χρειάζεται το πρόγραμμά μας υπάρχει στον δίσκο. Στην κύρια μνήμη παραχωρείται μια μικρότερη περιοχή (που μπορεί να είναι και *πολύ μικρότερη*). Κατά την εκτέλεση του προγράμματος, αναλόγως των αναγκών, φορτώνονται στην κύρια μνήμη κομμάτια της μνήμης από τον δίσκο ενώ φορτωμένα κομμάτια φυλάγονται στον δίσκο· έχουμε δηλαδή **ανταλλαγές μνήμης** (memory swapping) .◀

Συνεχίζουμε με το πρόγραμμά μας: Δίνουμε τιμές σε όλες τις δυναμικές μεταβλητές:

```
cin >> *a1;  *a2 = 4.56;  cin >> *k;
cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και βλέπουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Και τώρα πρόσεξε: Δίνουμε

```
delete a1;  delete a2;
cout << a1 << "  " << a2 << "  " << i << "  " << k << endl;
cout << *a1 << "  " << *a2 << "  " << *i << "  " << *k << endl;
```

και παίρνουμε:

```
0x3e2478  0x3e24e0  0x3e24f0  0x3e2500
1.23  4.56  375  1143
```

Παρά τις “**`delete`**” οι *a*1 και *a*2 κρατούν τις τιμές τους και οι αποπαραπομπές τους δεν έχουν το παραμικρό πρόβλημα! Και να ήταν μόνο αυτό... Δίνουμε στη συνέχεια τα εξής:

```
a1 = new double;
cout << a1 << "  " << *a1 << endl;
*a1 = 7.13;
cout << *a2 << endl;
```

και παίρνουμε:

```
0x3e24e0  4.56
```

[1] Έτσι, μπορεί να δείς και τον όρο «virtual address space»

7.13

Με την "a1 = new double" η *a1 υλοποιείται στην ίδια θέση με την *a2 (που όμως έχουμε «καταργήσει»). Και φυσικά όταν δίνουμε τιμή στην *a1 αλλάζουμε την τιμή (της καταργημένης) *a2.

Ας τα πάρουμε τώρα από την αρχή για να δούμε μερικά κρίσιμα «πρέπει» και «δεν πρέπει».

- *Δεν πρέπει να προσπαθήσεις να κάνεις αποπαραπομπή σε κάποιο βέλος αν δεν υπάρχει η αντίστοιχη δυναμική μεταβλητή-στόχος.* Πολύ περισσότερο, δεν πρέπει να προσπαθήσεις να αλλάξεις την τιμή της (ανύπαρκτης) δυναμικής μεταβλητής· αν η τυχαία τιμή που υπάρχει στο βέλος είναι «νόμιμη» διεύθυνση μπορεί να κάνεις ζημιά! Πώς αντιμετωπίζουμε πρακτικώς αυτό το πρόβλημα;
- *Σε κάθε μεταβλητή-βέλος που δηλώνεις πρέπει να δίνεις οπωσδήποτε αρχική τιμή. Αν δεν την ξέρεις δώσε τιμή* "0" ("NULL"). (Να υπενθυμίσουμε ότι στην §12.3.2 είδαμε και άλλους τρόπους –εκτός από μια παράσταση **new**– για να δώσουμε αρχική τιμή σε ένα βέλος. Εκεί είδαμε και την τιμή "0".)
- *Αν σε κάποιο σημείο του προγράμματός σου δεν έχεις τη σιγουριά ότι μια μεταβλητή-βέλος p δείχνει «νόμιμο» στόχο, πριν κάνεις αποπαραπομπή, πρέπει να ελέγξεις με μια* "if(p != 0) …"
- *Μετά από μια* "delete p", *αν δεν δίνεις μια άλλη νόμιμη τιμή στο βέλος p, πρέπει να βάλεις οπωσδήποτε μια* "p = 0" (για να έχει νόημα ο έλεγχος που υποδεικνύεται πιο πάνω). Η "delete p"
  - δεν μηδενίζει αυτομάτως την τιμή της *p*,
  - δεν «καθαρίζεται» η τιμή της δυναμικής μεταβλητής **p*· στο σημείο αυτό θα επανέλθουμε.[2]

16.4 Δυναμικοί Πίνακες

Το ενδιαφέρον της δυναμικής παραχώρησης μνήμης δεν βρίσκεται στο να πάρουμε μια μεταβλητή τύπου int όταν εκτελείται το πρόγραμμα αλλά

1. στη δυνατότητα υλοποίησης δυναμικών δομών στοιχείων (λίστες, δένδρα κλπ)
2. στη διαχείριση μεγάλων πινάκων.

Εδώ θα ασχοληθούμε με τη δεύτερη περίπτωση που χρειάζεται τη δεύτερη μορφή των **new** και **delete**. Έστω ότι στο πρόγραμμά μας δηλώνουμε:

```
double*  p( 0 );
```

Αν στη συνέχεια ζητήσουμε:

```
p = new double[100];
```

θα πάρουμε μια περιοχή μνήμης με 100 θέσεις τύπου double που μπορούμε να τις διαχειριστούμε σαν να είχαμε δηλώσει:

```
double p[100];
```

με την εξής διαφορά: αντί για το 100, θα μπορούσαμε να είχαμε βάλει στη new οποιαδήποτε παράσταση που θα μας έδινε μια θετική ακέραιη τιμή. Αυτή *η τιμή καθορίζεται τη στιγμή που εκετελείται η εντολή και όχι όταν μεταγλωττίζεται*, όπως γίνεται με τη δήλωση ενός συμβατικού πίνακα.

---

[2] Δηλαδή η "delete", παρά το όνομά της, δεν κάνει διαγραφές· απλώς παραδίδει στην *ανακύκλωση* τη δυναμική μνήμη που είχαμε σε χρήση.

Παρατήρηση: ▶
«Θετική ακέραιη τιμή»! Δηλαδή δεν μπορεί να είναι μηδέν; Η C++ δεν έχει αντίρρηση αλλά υπάρχουν κάτι προβληματάκια που μπορεί να μας έλθουν από τη C. Τα συζητούμε παρακάτω.◀

Χειριζόμαστε τον δυναμικό πίνακα όπως τους συμβατικούς. Για παράδειγμα τα στοιχεία του p είναι τα p[0], p[1], …, p[99] και χειριζόμαστε το κάθε ένα από αυτά όπως μια μεταβλητή τύπου `double`.

Όταν δεν μας χρειάζεται η μνήμη, μπορούμε να τη επιστρέψουμε με την εντολή:

```
delete[] p;
```

Αλλά εδώ υπάρχει κάτι που χρειάζεται ιδιαίτερη προσοχή: δεν επιτρέπεται να ανακατεύεις τις δύο μορφές των **new** και **delete**.

- ***Όταν παίρνεις δυναμική μνήμη με “`p = new T`” θα την επιστρέφεις με “`delete p`” και όταν παίρνεις δυναμική μνήμη με “`p = new T[...]`” θα την επιστρέφεις με “`delete[] p`”.***

Έτσι, *θα ήταν λάθος αν προσπαθούσαμε να επιστρέψουμε τον δυναμικό πίνακα* p *με μια* “`delete p`”. Αν πάλι είχαμε δηλώσει

```
double q( new double );
```

*είναι λάθος να προσπαθήσουμε να ανακυκλώσουμε την *q με* “`delete[] q`”.

Ιδιαίτερο ενδιαφέρον έχουν οι πολυδιάστατοι δυναμικοί πίνακες. Θα τα πούμε στη συνέχεια. Προς το παρόν ας ξαναδούμε ένα παράδειγμα από τα παλιά.

Παράδειγμα

Θα ξαναγράψουμε το πρόγραμμα που «μετράει» το αρχείο alturing.txt, που είδαμε στο Παράδ. 1 της §8.10, αλλά

- αφού φορτώσουμε το περιεχόμενο του αρχείου στη μνήμη (§15.12.1),
- σε δυναμικό πίνακα.

Κατ’ αρχάς θα γράψουμε μια

```
void loadText( string flNm, char*& toText, size_t& tLength )
```

που τροφοδοτείται με το όνομα ενός αρχείου (*flNm*), φορτώνει το περιεχόμενό του σε έναν δυναμικό πίνακα (με στοιχεία τύπου `char`) και μας επιστρέφει βέλος προς την αρχή του πίνακα (*toText*) και το πλήθος στοιχείων του πίνακα (*tLength*). Η συνάρτηση θα ρίχνει εξαίρεση αν δεν μπορεί να ανοίξει ή να διαβάσει το αρχείο ή αν δεν μπορεί να πάρει δυναμική μνήμη.

```
void loadText( string flNm, char*& toText, size_t& tLength )
{
   ifstream tin( flNm.c_str(), ios_base::binary|ios_base::ate );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotOpen,
                        flNm.c_str() );
   tLength = tin.tellg();
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
   tin.seekg( 0 );
   tin.read( toText, tLength );
   if ( tin.fail() )
      throw ApplicXptn( "loadText", ApplicXptn::cannotRead,
                        flNm.c_str() );
   tin.close();
   toText[tLength] = '\0';
} // loadText
```

Σε σχέση με αυτά που είδαμε στην §15.12.1 το νέο στοιχείο είναι η δυναμική μνήμη:

```
   try {  toText = new char[tLength+1];  }
   catch( bad_alloc& )
```

```
   {  throw ApplicXptn( "loadText", ApplicXptn::allocFailed );  }
```

Γιατί παίρνουμε *tLength*+1; Για να χωρέσει και ο φρουρός (`'\0'`) που βάζουμε στην τελευταία (παραπανίσια) θέση.

Σε μια άλλη συνάρτηση βάζουμε την επεξεργασία (μέτρημα):

```
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits )
{
   int pos( 0 );
   char ch;      // χαρακτήρας που διαβάζουμε
// Μηδένισε τους μετρητές
   nRows = 0;  nUpCase = 0;  nDigits = 0;
// Επεξεργάσου το αρχείο
   ch = toText[pos];
   while ( pos < tLength )
   {
      while ( pos < tLength && ch != '\n' )
      {
         if ( isupper(ch) ) ++nUpCase;
         else if ( isdigit(ch) )  ++nDigits;
         ++pos;  ch = toText[pos];
      }
      ++nRows;
      if ( pos < tLength )  {  ++pos;  ch = toText[pos];  }
   } // while
} // countText
```

Πρόσεξε τα εξής:

- Αν *pos* είναι ο δείκτης του χαρακτήρα *ch* που επεξεργαζόμαστε (toText`[`*pos*`] ==` *ch*) τότε η πρώτη "`t.get(ch)`" γίνεται: "`pos = 0; ch = toText[pos];`" και η επαναλαμβανόμενη: "`++pos; ch = toText[pos]`".
- Η "`!t.eof()`" γίνεται "`pos < tLength`". Θα μπορούσε να γίνει και "`ch != '\0'`"; Ναι, αλλά η πρώτη είναι προτιμότερη αφού δουλεύει και στην περίπτωση που υπάρχουν `'\0'` μέσα στο αρχείο. Να τονίσουμε, βέβαιως, ότι δεν περιμένουμε να βρούμε τέτοιους χαρακτήρες σε ένα αρχείο text.
- Ο έλεγχος τέλους γραμμής παραμένει: "`ch != '\n'`". Θα πρέπει να τον αλλάξουμε αν έχουμε αρχείο text που οι γραμμές του διαχωρίζονται από ακολουθία χαρακτήρων που δεν περιέχει τον `'\n'`.
- Αυτή η συνάρτηση δεν ρίχνει κάποια εξαίρεση; Θα μπορούσαμε να βάλουμε ελέγχους όπως "`toText == 0`" ή "`tLength < 0`" αλλά για ένα τόσο απλό πρόγραμμα δεν έχει νόημα.

Έχοντας δηλώσει:

```
   char*  toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t tLength;     // μήκος κειμένου
   int    nRows;       // μετρητής γραμμών
   int    nUpCase;     // μετρητής κεφαλαίων
   int    nDigits;     // μετρητής ψηφίων
```

καλούμε τις δύο συναρτήσεις ως εξής:

```
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
```

Ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <new>

using namespace std;
```

```
struct ApplicXptn
{
   enum { cannotOpen, cannotRead, allocFailed };
   char funcName[100];
   int  errCode;
   char errStrVal[100];

   ApplicXptn( const char* fn, int ec, const char* sv="" )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ApplicXptn

void loadText( string flNm, char*& toText, size_t& tLength );
void countText( const char* toText, int tLength,
                int& nRows, int& nUpCase, int& nDigits );

int main()
{
   char*     toText( 0 ); // βέλος προς ενταμιευτή κειμένου
   size_t    tLength;     // μήκος κειμένου
   int       nRows;       // μετρητής γραμμών
   int       nUpCase;     // μετρητής κεφαλαίων
   int       nDigits;     // μετρητής ψηφίων

   try
   {
      loadText( "alturing.txt", toText, tLength );
      countText( toText, tLength, nRows, nUpCase, nDigits );
      delete[] toText;  toText = 0;
   // Λέγε τα αποτελέσματα
      cout << " Διάβασα " << nRows << " γραμμές" << endl;
      cout << " Μέτρησα " << nUpCase << " κεφαλαία γράμματα και "
           << nDigits << " ψηφία" << endl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
         case ApplicXptn::cannotOpen:
            cout << "cannot open file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case ApplicXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal
                 << " in " << x.funcName << endl;
            break;
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

✎✎✎

16.5 * Η Τρίτη Μορφή του “new”

Υπάρχει και μια τρίτη μορφή χρήσης του τελεστή “**new**” που δεν έχει σχέση με δυναμική παραχώριση μνήμης. Δες ένα παράδειγμα: Οι εντολές

```
   int buf[100];
   double* d( new (buf) double );
   cout << &buf[0] << endl;
   cout << d << endl;
```

δίνουν:

```
0x22fde0
0x22fde0
```

Δηλαδή: η μεταβλητή **d* δημιουργείται στην αρχή του πίνακα buf.

Όπως φαίνεται και από το παράδειγμα, αυτή η μορφή του “**new**” έχει ως αποτέλεσμα *την δημιουργία μιας μεταβλητής σε μια συγκεκριμένη θέση της μνήμης*· δεν υπάρχει παραχώρηση δυναμικής μνήμης στο πρόγραμμά μας και επομένως δεν υπάρχει αντίστοιχη μορφή “**delete**”. Αυτή η μορφή λέγεται **new τοποθέτησης** (placement new).

Πρόσεξε ένα άλλο παράδειγμα:

```
   char* buf( new char[100] );
   double* d( new (buf) double );
// . . .
   delete[] buf;
```

Εδώ γιατί βάλαμε **delete**; Για να ανακυκλώσουμε τη buf που πήραμε με **new** *δεύτερης μορφής* (“**new char[100]**”). Φυσικά, ανακυκλώνεται και η **d*, αφού δημιουργήθηκε στην αρχή του πίνακα buf.

Σε μικρά μονοχρηστικά συστήματα μπορείς να χρησιμοποιήσεις αυτήν τη μορφή της “**new**” για να βάλεις μεταβλητές κατάλληλου τύπου σε συγκεκριμένες διευθύνσεις για να διαχειριστείς με το πρόγραμμά σου ενταμιευτές συσκευών ή σχετικούς καταχωρητές. Αν δουλεύεις σε Linux ή Windows ή Unix ή άλλο παρόμοιο σύστημα μη διανοηθείς να χρησιμοποιήσεις απόλυτες διευθύνσεις μνήμης. Όπως είπαμε και πιο πριν οι διευθύνσεις που βλέπει το πρόγραμμά σου είναι **εικονικές**.

16.6 Η Εξαίρεση *bad_alloc*

Οι σημερινοί υπολογιστές διαθέτουν τεράστιες ποσότητες μνήμης (σε σύγκριση με αυτά που ξέραμε μέχρι πριν από λίγα χρόνια). Με τα συστήματα εικονικής μνήμης τα όρια φτάνουν τη χωρητικότητα των δίσκων σου. Παρ’ όλα αυτά, ένα καλό πρόγραμμα θα πρέπει να είναι προετοιμασμένο για την περίπτωση που θα αποτύχει μια **new**, δηλαδή δεν θα μας δοθεί η μνήμη που ζητάμε. Στην περίπτωση αυτή ρίχνεται μια εξαίρεση τύπου *bad_alloc*.

Αυτό σημαίνει ότι η **new** θα πρέπει να βρίσκεται πάντοτε μέσα σε μια ομάδα **try** που θα ακολουθείται από μια “**catch(bad_alloc&)**”. Για να μπορέσεις να χρησιμοποιήσεις τη *bad_alloc* στο πρόγραμμά σου θα πρέπει να έχεις βάλει μια:

```
#include <new>
```

Κατ’ αρχήν λοιπόν έχουμε ένα σχέδιο της μορφής:

```
// . . .
#include <new>
// . . .
   try
   {
      double dr( new double[1000] );
// . . .
   }
// . . .
   catch( bad_alloc& )
   {
```

```
    // διαχείριση της εξαίρεσης
    }
// . . .
```

Στη συνέχεια θα βλέπεις να κάνουμε μια κάπως διαφορετική αντιμετώπιση: Θα πιάνουμε την εξαίρεση *bad_alloc* και θα ρίχνουμε μια δική μας,

```
// . . .
   double dr( 0 );
   try {  dr = new double[1000];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "thisFunc", MyProgXptn::allocFailed );  }
// . . .
```

Και τι θα μπορούσαμε να κάνουμε όταν πιάσουμε μια τέτοια εξαίρεση; Πιθανότατα να ανακυκλώσουμε κάποιον (ή κάποιους) τεράστιο πίνακα (-ες) που δεν μας χρειάζονται πια.

16.6.1 Μια Εξήγηση για τις Εξαιρέσεις μας

Τώρα μπορούμε να εξηγήσουμε και ένα χαρακτηριστικό των δικών μας εξαιρέσεων που μπορεί να σου φαίνεται περίεργο.

Γιατί δηλώνουμε

```
   char          funcName[100];
```

και όχι:

```
   string        funcName;
```

όπως συνήθως; Διότι ένα από τα προβλήματα που έχουμε να διαχειριστούμε είναι η έλλειψη δυναμικής μνήμης (*allocFailed*). Η *string* χρησιμοποιεί *δυναμική* μνήμη για να αποθηκεύσει το κείμενο. Αν λοιπόν το πρόβλημά μας είναι ότι *δεν μπορεί να παραχωρηθεί δυναμική μνήμη* και προσπαθήσουμε να ρίξουμε *εξαίρεση που θα χρειαστεί δυναμική μνήμη* –που πιθανότατα δεν θα μπορεί να πάρει– θα προκαλέσουμε «βίαιη» διακοπή της εκτέλεσης του προγράμματός μας. Χρησιμοποιώντας πίνακα τύπου `char` που υλοποιείται στη στοίβα τα πράγματα είναι ασφαλή.

16.7 Τα Προβλήματα της Δυναμικής Μνήμης

Η παραχώρηση δυναμικής μνήμης είναι ένα πολύ καλό εργαλείο αλλά έχει και ορισμένα προβλήματα η αντιμετώπιση των οποίων χρειάζεται την προσοχή μας όταν γράφουμε το πρόγραμμα.

Ξεκινάμε αντιγράφοντας –με ορισμένες προσαρμογές– ένα σχήμα και μερικά πράγματα από την §12.3.2:

Ας πούμε ότι δηλώνουμε:

```
   int* a( new int(10) );
   int* b( new int(20) );
```

Η εικόνα που θα έχουμε στη μνήμη είναι αυτή που βλέπεις στο Σχ. 16-3(α): το βέλος *a* δείχνει την **a*, που έχει τιμή 10, και το βέλος *b* δείχνει την **b* που έχει τιμή 20.

Έστω τώρα ότι εκτελείται η εντολή “`*a = *b`”. Η εικόνα της μνήμης είναι αυτή του Σχ. 16-3 (β). Η τιμή της **a* γίνεται 20, αλλά οι τιμές των βελών δεν αλλάζουν.

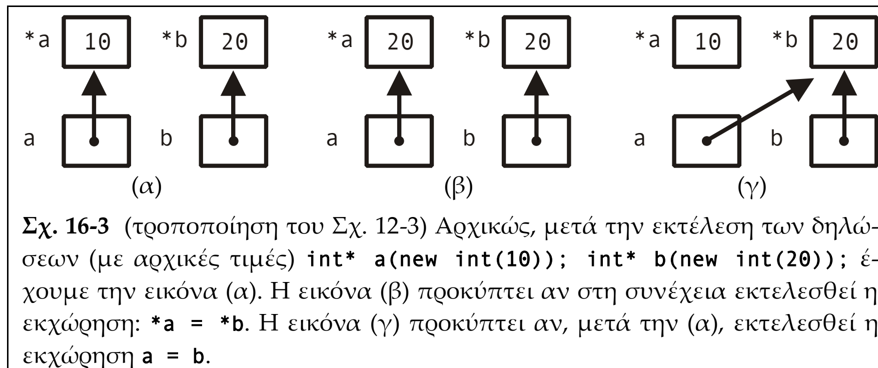
Ας δούμε όμως τι θα συμβεί αν, ενώ έχουμε την κατάσταση (α), εκτελεσθεί η εντολή: “`a = b`”. Οι τιμές των μεταβλητών **a* και **b* δεν αλλάζουν, αλλάζει όμως η τιμή της μεταβλητής *a*· το βέλος *a* δείχνει εκεί που δείχνει και το *b*: τη **b* (Σχ. 16-3 (γ)).

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

```
   cout << *a << "  " << *b << endl;
```

θα πάρουμε αποτέλεσμα:

```
20  20
```



Σχ. 16-3 (τροποποίηση του Σχ. 12-3) Αρχικώς, μετά την εκτέλεση των δηλώσεων (με αρχικές τιμές) `int* a(new int(10)); int* b(new int(20));` 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 `a = b`.

αλλά το «παρασκήνιο» είναι διαφορετικό· και στη δεύτερη περίπτωση δημιουργείται το εξής πρόβλημα: *το πρόγραμμά μας έχει μια δυναμική μεταβλητή, την* `*a`*, αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί* (*ούτε να την ανακυκλώσει*).

Αυτό που περιγράψαμε πιο πάνω, είναι γνωστό ως **απώλεια** ή **διαρροή μνήμης** (memory leakage). Μπορεί να εμφανισθεί στις εξής περιπτώσεις:

- Ένα βέλος *p* είναι το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης και χωρίς να την ανακυκλώσουμε, αλλάζουμε την τιμή του *p*, όπως είδαμε παραπάνω.
- Ένα βέλος *p*, όντας το μοναδικό που δείχνει μια περιοχή δυναμικής μνήμης, είναι τοπική μεταβλητή σε μια συνάρτηση. Η εκτέλεση της συνάρτησης τελειώνει χωρίς να ανακυκλώσουμε τη δυναμική μνήμη το βέλος *p* «χάνεται». Αυτό φαίνεται στο παρακάτω παράδειγμα.

Έστω ότι έχουμε τη συνάρτηση:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
} // f
```

Ας πούμε λοιπόν ότι

- καλείται η *f*,
- εκτελείται επιτυχώς η "`pi = new int[ni]`" και παίρνει μνήμη για τον pi, αλλά
- αποτυγχάνει να πάρει μνήμη για τον pd και ρίχνεται μια *MyProgXptn* από τη δεύτερη `throw`.
- Διακόπτεται η εκτέλεση της *f* και παύουν να υπάρχουν τα βέλη *pi* και *pd* (επιστρέφονται στη στοίβα).

Έτσι, το πρόγραμμά μας έχει δεσμεύσει μνήμη για *ni* θέσεις τύπου `int` αλλά δεν έχει πρόσβαση σε αυτήν. Τι θα έπρεπε να κάνουμε; Πριν ρίξουμε την εξαίρεση θα έπρεπε να ανακυκλώσουμε τη μνήμη που ήδη πήραμε:

```
// . . .
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
// . . .
```

Δεν τελειώσαμε όμως· αν μπορεί να εγερθεί εξαίρεση από αυτά που ακολουθούν θα πρέπει να ανακυκλώσουμε και τους δύο πίνακες:

```
void f( int ni, int nd, . . . )
{
   int*    pi( 0 );
   double* pd( 0 );

   try {  pi = new int[ni];  }
   catch( bad_alloc& )
   {  throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try {  pd = new double[nd];  }
   catch( bad_alloc& )
   {  delete[] pi;  // ανακύκλωσε τη μνήμη που ήδη πήρες
      throw MyProgXptn( "f", MyProgXptn::allocFailed );  }
   try
   {
   // άλλες εντολές
   }
   catch( ... )
   {
      delete[] pi;
      delete[] pd;
      throw;
   }
} // f
```

Πρόσεξε την τελευταία `catch`:

- πιάνει οποιαδήποτε εξαίρεση μπορεί να προέλθει από τις «*άλλες εντολές*»,
- ανακυκλώνει τους δύο πίνακες και
- την ξαναρίχνει!

Ένα άλλο πρόβλημα, από μια άποψη το «αντίστροφο» αυτού που είδαμε, είναι το πρόβλημα του **μετέωρου βέλους** (pending ή dangling pointer). Ας πούμε ότι είμαστε στην κατάσταση (γ) του Σχ. 16-3 και δίνουμε

```
   delete b;  b = 0;
```

Όπως βλέπεις, ανακυκλώνουμε τη **b* και –τηρώντας τον κανόνα που βάλαμε– μηδενίζουμε το βέλος *b*. Καλά όλα αυτά, αλλά η **b* ήταν ταυτοχρόνως και **a*· έτσι τώρα το βέλος *a* είναι *μετέωρο* αφού δείχνει σε θέση μνήμης που δεν ανήκει στο πρόγραμμά μας.

Και αυτό το πρόβλημα μπορεί να προέλθει από απρόσεκτο γράψιμο συναρτήσεων. Δες την παρακάτω:

```
void g( double* dAr, int n )
{
   int nn;
// . . .
   if ( n < nn )
   {
      double* dArL( new double[nn] );
// . . .
      delete[] dAr;  dAr = dArL;
      n = nn;
   } // if
// . . .
} // g
```

Αυτή η συνάρτηση τροφοδοτείται με έναν δυναμικό πίνακα. Σε κάποιο σημείο μπορεί να ανακαλύψει ότι το μέγεθος του πίνακα δεν είναι κατάλληλο και τον αλλάζει.

Προσοχή! Η παράμετρος-βέλος *dAr* είναι παράμετρος τιμής. Το ότι δεν έχουμε βάλει “`const`” μας επιτρέπει να αλλάζουμε τις τιμές των στοιχείων του πίνακα αλλά το *dAr* είναι αντίγραφο.της πραγματικής παραμέτρου· αν αλλάξουμε την τιμή του βέλους η αλλαγή είναι τοπική μέσα στη συνάρτηση.

Ας πούμε λοιπόν ότι καλούμε τη συνάρτηση ως εξής:

```
   double* bd( 0 );
   int bn( 0 );
// . . .
```

```
g( bd, bn );
```

Όταν αρχίσει η εκτέλεση της συνάρτησης το βέλος *dAr* είναι αντίγραφο του *bd* και έτσι δείχνουν και τα δύο την ίδια θέση της μνήμης. Αν η συνάρτηση βρει ότι το μέγεθος του πίνακα δεν είναι αρκετό παίρνει την κατάλληλη μνήμη (με το βέλος *dArL*) και ανακυκλώνει την παλιά ("`delete[] dAr`") –αφού, πιθανότατα, αντιγράψει το περιεχόμενό της.

- Στην περίπτωση αυτήν το *bd*, που έδειχνε τη μνήμη που ανακυκλώθηκε, είναι πια *μετέωρο*.
- Μετά το τέλος εκτέλεσης της συνάρτησης το *τοπικό* βέλος *dArL* χάνεται (επιστρέφει στη στοίβα) και δεν έχουμε εργαλείο για να χειριστούμε τη δυναμική μνήμη που έδειχνε. Έχουμε δηλαδή *διαρροή μνήμης*.

Το πρόβλημα εξαφανίζεται αν περάσουμε τον δυναμικό πίνακα με *παραμέτρους αναφοράς*:

```
void g( double*& dAr, int& n )
```

Τα επόμενα προβλήματα έχουν σχέση με τον τελεστή "`delete`".

Το πρώτο είναι αυτό που ήδη είπαμε (§16.4): *Η μορφή του "`delete`" με την οποίαν θα ανακυκλώνουμε τη δυναμική μνήμη κάθε φορά πρέπει να είναι αντίστοιχη της μορφής του "`new`" που χρησιμοποιήσαμε για την παραχώρησή της.* Αν παραβιάσεις αυτόν τον κανόνα μπορεί να κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης (heap).

Δεν υπάρχει κάποια «συνταγή» για να διασφαλίσεις την τήρηση αυτού του κανόνα. Η συνηθισμένη σχετική υπόδειξη λέει: *Για κάθε κομμάτι δυναμικής μνήμης που παίρνεις φρόντισε η παράσταση* **new** *και η αντίστοιχη παράσταση* **delete** *να βρίσκονται στην ίδια συνάρτηση.* Αν φροντίζεις να γράφεις και μικρές συναρτήσεις τότε κάτι μπορεί να γίνει... Πάντως, όπως θα καταλάβεις και από την πείρα σου, όταν γράφεις προγράμματα με δυναμική μνήμη, δεν είναι και τόσο απλό να τηρείς αυτόν τον κανόνα.

Ένα άλλο πρόβλημα με τη "`delete`" είναι το εξής: *αν*, κατά λάθος φυσικά, *αποπειραθείς να ανακυκλώσεις μνήμη που έχεις ήδη ανακυκλώσει* (και επομένως δεν ελέγχεται από το πρόγραμμά σου) *κάνεις ζημιά στο σύστημα διαχείρισης δυναμικής μνήμης*. Φυσικά, δεν λέει κανείς ότι θα πας να γράψεις στο πρόγραμμά σου

```
delete p;
delete p;
```

Είναι όμως πολύ πιθανό να γράψεις κάτι σαν:

```
if ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
while ( . . . )
{
// . . .
   delete p;
// . . .
}
delete p;
```

ή κάτι σαν:

```
f( ..., p, ... );
delete p;
```

και μέσα στην *f* εκτελείται άλλη μια "`delete p`".

Αν τηρείς τον κανόνα «*μηδένισε το βέλος μετά τη* `delete`» γλυτώνεις από αυτό το πρόβλημα. Αν έχουμε:

```
delete p;  p = 0;
```

```
delete p;
```

η δεύτερη "`delete p`" είναι στήν πραγματικότητα "`delete 0`" και

- ***Η "`delete 0`" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "`delete[] 0`".***

Το τελευταίο που έχουμε να πούμε είναι το εξής: *Μην προσπαθήσεις να ανακυκλώσεις μνήμη που δεν είναι δυναμική*. Θα δείξουμε πώς μπορεί να συμβεί κάτι τέτοιο με ένα παράδειγμα. Ας πούμε ότι θέλουμε να γράψουμε μια συνάρτηση που θα τροφοδοτείται με έναν πίνακα a με στοιχεία τύπου `int` και μια τιμή *v* τύπου `int`. Η συνάρτηση θα ψάχνει να βρει τη *v* μέσα στον a και αν δεν τη βρει θα την εισάγει. Τι θα κάνεις αν ο πίνακας είναι γεμάτος;

- Αν *ο πίνακας δεν είναι δυναμικός* η συνάρτηση θα πρέπει να βγάζει ένα μήνυμα –πιθανότατα μια εξαίρεση– για το πρόβλημα.
- Αν *ο πίνακας είναι δυναμικός* η συνάρτηση θα προσπαθήσει να τον «μεγαλώσει». Όπως θα δούμε στη συνέχεια, αυτό σημαίνει αντιγραφή σε έναν μεγαλύτερο πίνακα και ανακύκλωση του παλιού.

Όταν γράφεις τη συνάρτηση δεν ξέρεις με τι είδους πίνακες θα χρησιμοποιείται και αν ξεχάσεις την πρώτη απο τις παραπάνω δυνατότητες είναι πολύ πιθανό να κάνεις προσπάθεια να ανακυκλώσεις μη δυναμικό πίνακα.

Πώς λύνεται το πρόβλημα; Με διαχωρισμό των στόχων:

- Γράψε μια συνάρτηση που αναζητεί μια τιμή σε έναν πίνακα είτε αυτός είναι δυναμικός είτε όχι.
- Γράψε μια συνάρτηση που εισάγει μια τιμή σε έναν πίνακα που *δεν είναι γεμάτος* είτε αυτός είναι δυναμικός είτε όχι.

Αυτές μπορεί να είναι συναρτήσεις γενικής χρήσης· μπορεί να είναι και περιγράμματα.

Ανάμεσα στις κλήσεις αυτών των δύο συναρτήσεων βάλε τις εντολές –που εξαρτώνται από το πρόβλημα– για τις κατάλληλες ενέργειες όταν ο πίνακας είναι γεμάτος.

Αργότερα, όταν θα δούμε την STL, θα δούμε ότι υπάρχουν εργαλεία που μας απαλλάσσουν από αυτά τα προβλήματα. Τέτοια είναι:

- τα **έξυπνα βέλη** (smart pointers) που δεν έχουν αυτά τα προβλήματα και
- το *vector* και τα άλλα περιγράμματα **περιεχόντων** (containers).

16.7.1 RAII: Μια Καλύτερη Λύση

Τώρα, θα ξαναγυρίσούμε στη συνάρτηση *f* που είδαμε ως παράδειγμα για τη διαρροή μνήμης μέσα σε συνάρτηση· ναι αυτήν με τις τρεις **try-catch**! Θα τη χρησιμοποιήσουμε τώρα ως παράδειγμα για να παρουσιάσουμε μια άλλη λύση, σαφώς καλύτερη, με μια πάγια τεχνική της C++ που ονομάζεται «*Resource Acquisition Is Initialization*» (RAII), δηλαδή: *πρόσκτηση πόρων σημαίνει εκκίνηση*.

Πριν προχωρήσεις καλό θα είναι να κάνεις μια επανάληψη σε όσα είπαμε –έστω και ακροθιγώς– για δημιουργούς (§15.3) και καταστροφείς (§15.3.1).

Για το παράδειγμά μας τώρα, ορίζουμε έναν τύπο ως εξής:

```
struct IntDArr
{
   int* da;
   IntDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "IntDArr", MyProgXptn::allocFailed );  }
   }
   ~IntDArr()  {  delete[] da;  }
}; // IntDArr
```

Κάθε μεταβλητή αυτού του τύπου «κρύβει» μέσα της έναν δυναμικό πίνακα με στοιχεία τύπου `int`. Μέσα στη συνάρτηση, αντί για “`int* pi(0)`” βάζουμε τη δήλωση:

```
   IntDArr    pi( ni );
```

Για την εκτέλεσή της καλείται ο δημιουργός που παίρνει τη μνήμη που απαιτείται για τον δυναμικό πίνακα `pi.da`.

Σημείωση: ▶
Εδώ όμως έχουμε και καινούρια πράγματα: *Εξαίρεση από δημιουργό*! Ναι! Αν κάτι «πάει στραβά» (δεν πήραμε μνήμη) ρίχνεται εξαίρεση και *δεν δημιουργείται το αντικείμενο*!

Αυτό βέβαια μας βάζει και ιδέες· δεν θα μπορούσαμε να βάλουμε ελέγχους στον δημιουργό της *Date* –ας πούμε– και να ρίχνουμε εξαίρεση αν μας δώσουν μήνα 37; Ναι και θα το κάνουμε αργότερα. ◀

Στη συνέχεια, μπορούμε να χρησιμοποιούμε τον πίνακα, αλλά για το στοιχείο *k* θα πρέπει να γράφουμε `pi.da[k]` αντί για `pi[k]`. Αν διακοπεί η εκτέλεση της συνάρτησης για κάποιο λόγο –είτε διότι φτάσαμε στο τέλος της είτε λόγω εξαίρεσης– η μεταβλητή `pi` θα καταστραφεί με *αυτόματη κλήση του καταστροφέα*. Ο καταστροφέας, όπως βλέπεις, ανακυκλώνει τον πίνακα και έτσι δεν έχουμε διαρροή μνήμης.

Πολύ ωραία, αλλά για να δούμε την εξής περίπτωση: Ας πούμε ότι κάποια από τις παραμέτρους της συνάρτησης που δεν βλέπουμε είναι “`int*& ipp`” και –αν όλα πάνε καλά– θέλουμε να δείχνει στον νέο πίνακα (που έτσι θα «επιζήσει» μετά την εκτέλεση της συνάρτησης). Κανένα πρόβλημα:

```
   int* psv( pi.da );
   pi.da = ipp;
   ipp = psv;
```

Δηλαδή: αντιμεταθέτουμε τις τιμές των βελών `pi.da` και `ipp` και έτσι το `ipp` δείχνει τον νέο δυναμικό πίνακα ενώ, όταν καταστραφεί η `pi`, θα ανακυκλωθεί ο πίνακας που έδειχνε αρχικώς η `ipp` (αν δεν είχε τιμή 0).

Για να χειριστούμε παρομοίως και τον δυναμικό πίνακα τύπου `double` ορίζουμε:

```
struct DoubleDArr
{
   int* da;
   DoubleDArr( int n )
   {
      try {  da = new int[n];  }
      catch( bad_alloc& )
      {  throw MyProgXptn( "DoubleDArr", MyProgXptn::allocFailed ); }
   }
   ~DoubleDArr()  {  delete[] da;  }
}; // DoubleDArr
```

και δες πώς γίνεται η *f*:

```
void f( int ni, int nd, . . . )
{
   IntDArr    pi( ni );
   DoubleDArr pd( nd );

// . . . όπως πριν αλλά
// αντικαθιστώντας το κάθε pi[k] με pi.da[k] και
// το κάθε pd[k] με pd.da[k]
} // f
```

Αγνώριστη και πολύ καλύτερη!

Αργότερα, όταν εξοικειωθείς περισσότερο με δημιουργούς και καταστροφείς θα κατάνοήσεις καλύτερα την τεχνική και θα καταλάβεις ότι είναι πολύτιμη σε πολλές περιπτώσεις.

Οι *IntDArr* και *DoubleDArr* λέγονται **δομές** (ή **κλάσεις**) **περιτυλίγματος** (wrapper structures ή classes). Θα τις ξαναδούμε και για άλλες δουλειές.

16.8 Προβλήματα και στις Δομές

Τώρα θα επισημάνουμε άλλο ένα πρόβλημα αλλά θα αφήσουμε τη λύση του για αργότερα.

Ας πούμε ότι κάνουμε τις εξής αλλαγές στον τύπο *Address*:

```
struct AddressD
{
   char* country;
   char* city;
   int   areaCode;
   char* street;
   int   number;
}; // AddressD
```

με σκοπό να κάνουμε οικονομία και να μην σπαταλούμε τον ίδιο χώρο για την οδό "Κώ" και την οδό "Αγίου Κωνσταντίνου". Όλα δυναμικά!

Πρόσεξε τώρα: ας πούμε ότι έχουμε:

```
   AddressD a1, a2;
```

και –αφού πάρουμε την απαραίτητη δυναμική μνήμη– δίνουμε τιμές σε όλα τα μέλη του *a*1.

Τι αποτέλεσμα θα έχει η εκχώρηση "`a2 = a1`" στη συνεχεια; Τα τρία *βέλη a*1*.country*, *a*1*.city*, *a*1*.street θα αντιγραφούν στα αντίστοιχα βέλη* του a2 *αλλά δεν θα έχουμε αντιγραφές κειμένων*. Έτσι, θα έχουμε τρεις δυναμικούς πίνακες χαρακτήρων που ο καθένας του στοχεύεται από δύο βέλη. Αλλάζουμε την πόλη στο *a*1 αλλάζει και η πόλη στο *a*2· αλλάζουμε τη χώρα στο *a*2 αλλάζει και η χώρα στο *a*1. Σπανίως θέλουμε να συμβαίνει κάτι τέτοιο.

Η διόρθωση αυτής της συμπεριφοράς γίνεται με τη σωστή επιφόρτωση του τελεστή εκχώρησης. Αυτό θα το δούμε αργότερα.[3] Προς το παρόν, αν θέλεις να μην σπαταλάς μνήμη και να κάνεις εύκολα τη δουλειά σου, τη λύση τη ξέρεις: χρησιμοποίησε τον τύπο *string*.[4]

16.9 Δισδιάστατοι Δυναμικοί Πίνακες

Θα δούμε τώρα πώς μπορούμε να έχουμε δισδιάστατους δυναμικούς πίνακες.

Ας πούμε ότι μας δίνεται ένα μορφοποιημένο αρχείο (όνομα στον δίσκο `egr63e2.txt`) όπου υπάρχουν, σίγουρα, οι τιμές των στοιχείων (πραγματικοί) ενός δισδιάστατου πίνακα. Στην πρώτη γραμμή υπάρχουν δύο θετικοί ακέραιοι που δίνουν το πλήθος γραμμών και το πλήθος στηλών του πίνακα. Σε κάθε μια από τις επόμενες γραμμές του αρχείου υπάρχουν οι τιμές των στοιχείων μιας γραμμής του πίνακα. Θέλουμε να διαβάσουμε το αρχείο και να αποθηκεύσουμε το περιεχόμενό του σε έναν δυναμικό δισιδιάστατο πίνακα.

Αρχίζουμε ανοίγοντας το αρχείο. Αμέσως μετά διαβάζουμε τους αριθμούς γραμμών και στηλών:

```
   ifstream tin( "egr63e2.txt" );
   int      nR, nC;
   tin >> nR >> nC;
```

Ο πίνακάς μας θα είναι $nR \times nC$.

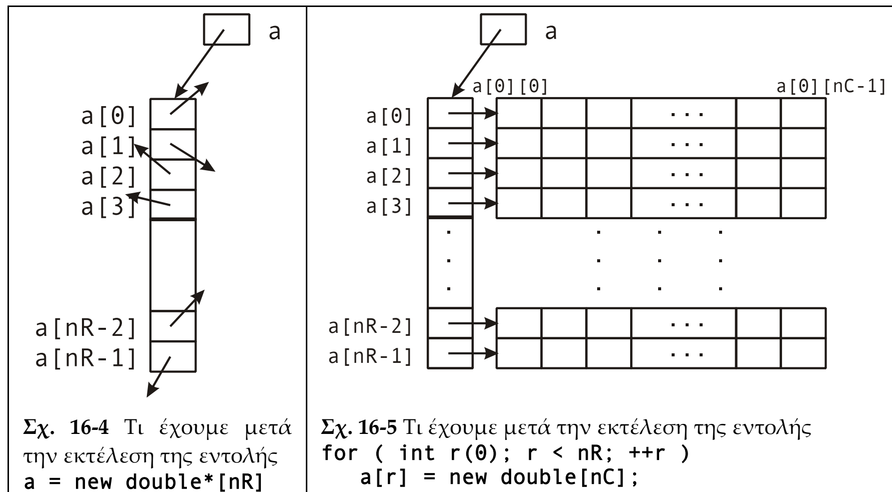
Αν δούμε τον δισδιάστατο δυναμικό πίνακα ως *πίνακα γραμμών* –*πίνακα πινάκων*– θα έχουμε *nR* μονοδιάστατους δυναμικούς πίνακες που καθένας τους θα έχει *nC* στοιχεία τύπου `double`. Για κάθε έναν από αυτούς τους πίνακες χρειαζόμαστε ένα βέλος τύπου `double*`· επομένως θα χρειαστούμε έναν δυναμικό πίνακα με *nR* στοιχεία τύπου `double*`:

```
   (double*)* a;
```

---

[3] Μην προσπαθήσεις να επιφορτώσεις τον "`=`" για τύπο δομής με αυτά που είπαμε στην §14.6.3. Αργότερα θα μάθουμε πώς γίνεται.

[4] Ε, δεν μπορεί να τρόμαξες από αυτά που είπαμε για τη *save()* και τη *load()*! Αφού, όπως είδες, όλα διορθώνονται.



Σχ. 16-4 Τι έχουμε μετά την εκτέλεση της εντολής `a = new double*[nR]`

Σχ. 16-5 Τι έχουμε μετά την εκτέλεση της εντολής
```
for ( int r(0); r < nR; ++r )
    a[r] = new double[nC];
```

ή απλούστερα:

```
    double** a;
```

Έχουμε δηλαδή ένα *διπλό βέλος* ή *βέλος προς βέλος*.

Μόλις μάθουμε το πλήθος γραμμών nR του πίνακα μπορούμε να πάρουμε μνήμη για αυτόν τον πίνακα βελών:

```
    a = new double*[nR];
```

Στο Σχ. 16-4 βλέπεις μια εικόνα του πίνακα a μετά την εκτέλεση της παραπάνω εντολής. Τα στοιχεία (βέλη) του πίνακα, τα `a[`r`]`, r: `0..`nR-1, δεν είναι ορισμένα· αυτό παριστάνεται με τα βέλη που δείχνουν σε τυχαίες διευθύνσεις, νόμιμες ή όχι.

Το τελευταίο βήμα είναι η υλοποίηση των γραμμών με δυναμική μνήμη. Αυτό γίνεται με την:

```
    for ( int r(0); r < nR; ++r ) a[r] = new double[nC];
```

Τη μορφή του πίνακα μετά από αυτήν την εντολή τη βλέπεις στο Σχ. 16-5. Το στοιχείο c της γραμμής r είναι το "`(a[r])[c]`" ή απλούστερα "`a[r][c]`". Αυτόν τον συμβολισμό τον ξέρουμε ήδη από τους μη δυναμικούς δισδιάστατους πίνακες.

Αν θελήσουμε να διαβάσουμε τα στοιχεία του πίνακα από το αρχείο, αυτό θα γίνει κατά τα γνωστά:

```
    for ( int r(0); r < nR; ++r )
    {
        for ( int c(0); c < nC; ++c ) tin >> a[r][c];
    }
    tin.close();
```

Σε όλα αυτά που είπαμε δεν βάλαμε ελέγχους για να αναδείξουμε αυτά που μας ενδιαφέρουν εδώ. Στη συνέχεια θα τα ξαναδείς με τους ελέγχους τους (τους ελέγχους για την ανάγνωση από το αρχείο τους ξέρεις καλά.)

Και πώς ανακυκλώνουμε τη μνήμη όταν δεν τη χρειαζόμαστε; Πρώτα ανακυκλώνουμε τις γραμμές:

```
    for ( int r(0); r < nR; ++r ) delete[] a[r];
```

και μετά τον πίνακα των βελών:

```
    delete[] a;
```

Αφού όλα αυτά διαφέρουν μόνον ως προς τον τύπο των στοιχείων του πίνακα μπορούμε να τα βάλουμε σε δύο περιγράμματα. Εδώ θα βάλουμε και ελέγχους και καλό είναι να τους προσέξεις. Θα χρησιμοποιήσουμε μια δομή εξαιρέσεων που είναι κάτι σαν:

```
struct MyTmpltLibXptn
{
    enum {  domainError, noArray, allocFailed  };

    char funcName[100];
```

```
   int  errCode;
   int  errVal1, errVal2;
   MyTmpltLibXptn( const char* fn, int ec,
                   int erv1 = 0, int erv2 = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errVal1 = erv1;  errVal2 = erv2;  }
}; // MyTmpltLibXptn
```

Θα τη δούμε σε επόμενο κεφάλαιο.

Το πρώτο περίγραμμα –το ονομάζουμε *new2d*– τροφοδοτείται με τα πλήθη γραμμών και στηλών και επιστρέφει βέλος τύπου *T*** (*T* ο τύπος-παράμετρος):

```
  0: template< typename T >
  1: T** new2d( int nR, int nC )
  2: {
  3:    if ( nR <= 0 || nC <= 0 )
  4:       throw MyTmpltLibXptn( "new2d",
  5:                             MyTmpltLibXptn::domainError,
  6:                             nR, nC );
  7:    T** fv;
  8:    try {  fv = new T*[nR];  }
  9:    catch( bad_alloc& )
 10:    {  throw MyTmpltLibXptn( "new2d",
 11:                             MyTmpltLibXptn::allocFailed );  }
 12:    for ( int r(0); r < nR; ++r )
 13:    {
 14:       try {  fv[r] = new T[nC];  }
 15:       catch( bad_alloc& )
 16:       {
 17:          for ( int k(0); k < r; ++k )  delete[] fv[k];
 18:          delete[] fv;
 19:          throw MyTmpltLibXptn( "new2d",
 20:                                MyTmpltLibXptn::allocFailed );
 21:       } // catch
 22:    } // for ( int r
 23:    return fv;
 24: } // new2d
```

Στη γρ. 8 προσπαθούμε να πάρουμε μνήμη για τα βέλη των γραμμών· αν δεν τα καταφέρουμε ρίχνουμε εξαίρεση και τελειώσαμε. Στη γρ. 14, που είναι μέσα στην περιοχή επανάληψης της **for** (γρ. 12) προσπαθούμε να πάρουμε μνήμη για τη γραμμή *r*. Μέχρι το σημείο αυτό έχουμε πάρει ήδη μνήμη για τα βέλη των γραμμών αλλά και για τις γραμμές 0 .. *r*-1. Αν δεν μπορέσουμε να πάρουμε μνήμη για τη γραμμή *r* πρέπει –πριν ρίξουμε την εξαίρεση– να ανακυκλώσουμε όλη αυτή τη μνήμη που ήδη πήραμε. Αυτό ακριβώς κάνουμε στις γρ. 17 και 18.

Η *delete2d()* είναι πολύ πιο απλή:

```
template< typename T >
void delete2d( T**& a, int nR )
{
   if ( a == 0 && nR > 0 )
      throw MyTmpltLibXptn( "delete2d", MyTmpltLibXptn::noArray );
   for ( int r(0); r < nR; ++r )  delete[] a[r];
   delete[] a;  a = 0;
} // delete2d
```

Πρόσεξε μόνον τον τύπο της πρώτης παραμέτρου: "`T**&`" διπλό βέλος αναφοράς! Εντυπωσιακό μεν αλλά απολύτως φυσικό: το a είναι ένα διπλό βέλος του οποίου η τιμή αλλάζει μέσα στη συνάρτηση.

Πρόσεξε ότι κάνουμε και αυτό που δεν κάνει η C++: μηδενίζουμε το βέλος μετά την ανακύκλωση του στόχου του.

Με αυτά τα εργαλεία, το παράδειγμα που ξεκινήσαμε γράφεται:

```
   tin >> nR >> nC;
   a = new2d<double>( nR, nC );
```

```
   for ( int r(0); r < nR; ++r )
   {
      for ( int c(0); c < nC; ++c ) tin >> a[r][c];
   }
   tin.close();
//  . . .
   delete2d( a, nR );
```

Πρόσεξε ότι η *new2d()* δεν έχει κάποια παράμετρο με βάση την οποία θα μπορούσε να γίνει αυτόματη εξειδίκευση. Έτσι η κλήση θα πρέπει να γίνει υποχρεωτικώς ως: "`new2d<double>`".

Για να συγκρίνεις τους δυναμικούς δισδιάστατους πίνακες με τους συμβατικούς θα γράψουμε για τέταρτη (!) φορά το πρόγραμμα πολλαπλασιασμού πινάκων που πρωτοείδαμε στο Παράδ 4 της §12.4 (χωρίς συναρτήσεις), ξαναείδαμε στο Παράδ. 6 της §13.9.3 (συνάρτηση με παράμετρο τον «γραμμοποιημένο» πίνακα) και τέλος στην §14.7.1, όπου δείξαμε και ένα τέχνασμα με τις παραμέτρους περιγραμμάτων.

Τώρα θα το δούμε με δυναμικούς πίνακες:

```
   const int l( 3 ),  m( 5 ),  n( 2 );

   int** a( new2d<int>(l, m) );
   int** b( new2d<int>(m, n) );
   int** d( new2d<int>(l, n) );
```

Πρόσεξε πώς γράφονται τώρα οι συναρτήσεις εισόδου/εξόδου των πινάκων:

```
void input2DAr( istream& tin, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
      for ( int c(0); c < nCol; ++c )
         tin >> a[r][c];
} // input2Dar

void output2DAr( ostream& tout, int** a, int nRow, int nCol )
{
   for ( int r(0); r < nRow; ++r )
   {
      for ( int c(0); c < nCol; ++c )
      {
         tout.width(3);  cout << a[r][c] << "  ";
      }
      cout << endl;
   }
} // output2DAr
```

Ο πίνακας περνάει στη συνάρτηση με: (διπλό) βέλος, πλήθος γραμμών, πλήθος στηλών. Μέσα στις συναρτήσεις χρησιμοποιείται με τον συνήθη συμβολισμό των δισδιάστατων πινάκων (χωρίς «γραμμοποιήσεις»).

Οι συναρτήσεις καλούνται ως εξής: Διαβάζουμε με τις:

```
   input2DAr( atx, a, l, m );
   input2DAr( atx, b, m, n );
```

και γράφουμε με τις:

```
   cout << " Στοιχεία του πίνακα a" << endl;
   output2DAr( cout, a, l, m );
   cout << " Στοιχεία του πίνακα b" << endl;
   output2DAr( cout, b, m, n );
   cout << " Στοιχεία του πίνακα d" << endl;
   output2DAr( cout, d, l, n );
```

Στο τέλος, ανακυκλώνουμε τη μνήμη που πήραμε με τις:

```
   delete2d( a, l );
   delete2d( b, m );
   delete2d( d, l );
```

Για όποιον δεν το πρόσεξε να το επισημάνουμε: *Οι γραμμές ενός δισδιάστατου δυναμικού πίνακα δεν είναι απαραίτητο να έχουν το ίδιο μήκος!* Ας δούμε ένα

Παράδειγμα

Θέλουμε έναν δυναμικό δισδιάστατο πίνακα που κάθε γραμμή του θα έχει το όνομα ενός μήνα. Δηλώνουμε:

```
   char** monthName( 0 );
```

και παίρνουμε μνήμη ως εξής:

```
   monthName = new char*[12];

   monthName[0] = new char[strlen("January")+1];
   strcpy( monthName[0], "January" );
   monthName[1] = new char[strlen("February")+1];
   strcpy( monthName[1], "February" );
// . . .
   monthName[10] = new char[strlen("November")+1];
   strcpy( monthName[10], "November" );
   monthName[11] = new char[strlen("December")+1];
   strcpy( monthName[11], "December" );
```

Όπως βλέπεις, για κάθε μήνα παίρνουμε ακριβώς τη μνήμη που χρειαζόμαστε (το “+1” για τον φρουρό).

Όταν θέλουμε να ανακυκλώσουμε τη μνήμη, η

```
   delete2d( monthName, 12 );
```

κάνει τη δουλειά μια χαρά.

Παρατήρηση: ▶

Αυτά για το παράδειγμα. Σε πραγματικές καταστάσεις. πολύ πιο βολική λύση είναι η

```
   string* sMonthName( new string[12] );
// . . .
   delete[] sMonthName;
```

Αν πάρουμε υπόψη μας ότι σε μια τιμή *string* το κείμενο αποθηκεύεται σε δυναμική μνήμη, έχουμε και πάλι δυναμικό πίνακα δυναμικών πινάκων.◀

16.10 * Τύπος Βέλους: “void*”

Η C δεν υποστηρίζει περιγράμματα συναρτήσεων και για να διευκολύνει τους προγραμματιστές στην προσπάθειά τους να γράψουν συναρτήσεις ανεξάρτητες από τύπο (δίνουμε ένα παράδειγμα στη συνέχεια) δίνει και άλλο ένα εργαλείο: τον τύπο γενικού βέλους “void*”.

Σε ένα βέλος αυτού του τύπου μπορείς να δώσεις ως τιμή τη διεύθυνση αντικειμένου οποιοδήποτε τύπου. Για παράδειγμα:

```
   int iv;
   void* pv1( &iv );
   double dv;
   pv1 = &dv;
   Date dt;
   pv1 = &dt;
```

Η αντίστροφη εκχώρηση δεν μπορεί να γίνει χωρίς τυποθεώρηση, π.χ.:

```
   int* pInt( reinterpret_cast<int*>(pv1) );
```

Στα βέλη **void*** δεν μπορούμε να κάνουμε αποπαραπομπή αν προηγουμένως δεν κάνουμε ερμηνευτική τυποθεώρηση (`reinterpret_cast`).

Αναφέραμε τα παραπάνω μόνο για προετοιμασία για την επόμενη παράγραφο και δεν θα δεις να τα χρησιμοποιούμε ξανά.

Ας δούμε ένα παράδειγμα γενικής συνάρτησης της C ώστε να δεις και χρήση του "`void*`" αλλιώτικο από αυτό που θα δεις στην επόμενη παράγραφο. Βάζοντας στο πρόγραμμά σου "`#include <cstdlib>`" μπορείς να χρησιμοποιήσεις τη συνάρτηση της C

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

που ταξινομεί –με τον αλγόριθμο Quicksort– *num* στοιχεία ενός πίνακα ξεκινώντας από αυτό που δείχνει το βέλος *base*. Η παράμετρος *size* περνάει το μέγεθος (σε ψηφιολέξεις) ενός στοιχείου του πίνακα. Η τελευταία παράμετρος είναι (βέλος προς) μια συνάρτηση. Όταν καλείται από την *qsort()* επιστρέφει:

- *Αρνητική* τιμή αν το στοιχείο που δείχνει η *πρώτη* παράμετρος *προηγείται* του στοιχείου που δείχνει η δεύτερη παράμετρος.
- *Μηδέν* αν το στοιχείο που δείχνει η *πρώτη* παράμετρος είναι *ίσο* με το στοιχείο που δείχνει η δεύτερη παράμετρος.
- *Θετική* τιμή αν το στοιχείο που δείχνει η *δεύτερη* παράμετρος *προηγείται* του στοιχείου που δείχνει η πρώτη παράμετρος.

Εδώ βλέπεις έναν άλλον τρόπο να γράφεις γενικές συναρτήσεις: με διευθύνσεις (βέλη) και μεγέθη περιοχών στη μνήμη. Έχουμε δηλαδή προγραμματισμό χαμηλού επιπέδου, που, όπως έχουμε πει, είναι δύσκολος. Πάντως υπάρχει και ένα (μικρό) κέρδος: ενώ, όπως είπαμε, ο μεταγλωττιστής θα δημιουργήσει μια συνάρτηση για κάθε στιγμιότυπο του περιγράμματος που θα βρει στο πρόγραμμά μας, η *qsort()* –και οποιαδήποτε συνάρτηση γραμμένη με αυτόν τον τρόπο– θα μεταγλωττισθεί μια φορά μόνο. Για κάθε κλήση της θα πρέπει να γράψουμε μια (το πολύ) συνάρτηση *compar()*.

Ας πούμε λοιπόν ότι σε κάποιο πρόγραμμα έχουμε:

```
int    intArr[ 50 ];
double dblArr[ 100 ];
```

και θέλουμε να ταξινομήσουμε ολόκληρον τον dblArr, κατ' *αύξουσα* τάξη, με χρήση της *qsort()*. Θα την καλέσουμε ως εξής:

```
qsort( dblArr, 100, sizeof(double), compareDblInc );
```

Το πρώτο όρισμα θα μπορούσε, αντί για "`dblArr`", να είναι "`&dblArr[0]`". Σε κάθε περίπτωση αυτό αντιγράφεται στο *base*. Πώς θα είναι η *compareDblInc()*; Έτσι:[5]

```
int compareDblInc( const void* a, const void* b )
{
   return ( *reinterpret_cast<const double*>(a) -
            *reinterpret_cast<const double*>(b) );
} // compareDblInc
```

Αφού θέλουμε ταξινόμηση κατ' αύξουσα τάξη το "`7.1`" προηγείται του "`11.7`". Έτσι η κλήση "`compareDblInc(&dblArr[35],&dblArr[71])`", όπου τα δύο στοιχεία έχουν αντιστοίχως τις παραπάνω τιμές, θα επιστρέψει τιμή "`-4`" (`== int(-4.6)`). Η *qsort()* θα κάνει πολλές τέτοιες κλήσεις.

Αν θέλουμε να ταξινομήσουμε το κομμάτι `intArr[11]` μέχρι `intArr[20]` κατά *φθίνουσα* τάξη θα πρέπει να γράψουμε:

```
qsort( &intArr[11], 10, sizeof(int), compareIntDec );
```

όπου:

```
int compareIntDec( const void* a, const void* b )
{
   return ( *reinterpret_cast<const int*>(b) -
            *reinterpret_cast<const int*>(a) );
} // compareIntDec
```

Πρόσεξε ότι με αυτήν το "`11`" προηγείται του "`7`".

[5] Στη C θα γράφανε: "`return ( *(double*)a - *(double*)b );`".

16.11 * Αναμνήσεις από τη C: *malloc()*, *free()*, *realloc()*

Η C++, έχοντας τη C ως υποσύνολό της, δίνει τη δυνατοτητα διαχείρισης της δυναμικής μνήμης και με τις συναρτήσεις *malloc()* (*calloc()*), *free()* και *realloc()*. Επειδή είναι πολύ πιθανό να τις συναντήσεις σε προγράμματα αξίζει να τις δούμε, έστω και εν συντομία. Για να τις χρησιμοποιήσεις θα πρέπει να περιλάβεις στο πρόγραμμά σου το `cstdlib`.

Η επικεφαλίδα της *malloc()* είναι:

```
void* malloc( size_t size );
```

Το “`void*`” λέει ότι η συνάρτηση επιστρέφει βέλος χωρίς τύπο. Μέσω της παραμέτρου πρέπει να περάσουμε την ποσότητα της ζητούμενης μνήμης σε ψηφιολέξεις. Αν έχεις δηλώσει:

```
int* pInt;
```

και ζητήσεις:

```
pInt = reinterpret_cast<int*>( malloc(72*sizeof(int) );
```

θα σου παραχωρηθεί που χωράει 72 τιμές τύπου `int` (πίνακας με 72 στοιχεία τύπου `int`) Το βέλος *pInt* δείχνει την αρχή της μνήμης που παραχωρήθηκε. Αν δεν υπάρχει διαθέσιμη η μνήμη που ζητείται η τιμή που επιστρέφεται είναι 0 (μηδέν).

Δεν είναι παράνομο να καλέσεις τη *malloc()* με όρισμα 0 αλλά το αποτέλεσμα εξαρτάται από την εγκατάσταση:

- Μπορεί να σου επιστρέψει βέλος 0.
- Μπορεί να σου επιστρέψει μη μηδενικό βέλος στο οποίο όμως δεν μπορείς να κάνεις αποπαραπομπή.

Παρόμοια με τη *malloc()* είναι η:

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

Αυτή ταιριάζει πιο πολύ σε πίνακες: σου επιτρέπει να ζητήσεις *nItems* θέσεις μνήμης που κάθε μια τους θα πιάνει *size* ψηφιολέξεις. Το παραπάνω παράδειγμα θα μπορούσε να γραφεί:

```
pInt = reinterpret_cast<int*>( calloc(72, sizeof(int) );
```

Όπως καταλαβαίνεις, αυτές οι δύο συναρτήσεις παίζουν τον ρόλο του τελεστή `new`.

Τον ρόλο του `delete` τον παίζει η συνάρτηση

```
void free( void* block );
```

που επιστρέφει στον σωρό τη μνήμη που δείχνει το βέλος *block*.

Πολύ συχνά, όταν δουλεύουμε με δυναμικούς πίνακες, έχουμε ανάγκη να μεγαλώσουμε (ή και να μικρύνουμε) κάποιον τον πίνακα. Για την περίπτωση αυτήν η C++ (C) μας δίνει τη συνάρτηση

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

που επεκτείνει ή συρρικνώνει το κομμάτι δυναμικής μνήμης που δείχνει το βέλος *block* σε *size* ψηφιολέξεις. Συνήθως αυτό γίνεται με παραχώρηση νέου τμήματος μνήμης στο οποίο η *realloc()* φροντίζει να αντιγράψει το περιεχόμενο του αρχικού τμήματος.

Το παρακάτω παράδειγμα δείχνει τη χρήση των συναρτησεων.

```
 0: #include <iostream>
 1: #include <cstdlib>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    int*    ip( 0 );
 8:    double* dp( 0 );
 9:
10:    ip = reinterpret_cast<int*>( malloc(3*sizeof(int)) );
11:    dp = reinterpret_cast<double*>( calloc(3,sizeof(double)));
12:    for ( int k(0); k <= 2; ++k )
```

```
13:    {
14:       ip[k] = k+1;
15:       dp[k] = 1.1*ip[k];
16:    }
17:    for ( int k(0); k <= 2; ++k )
18:       cout << ip[k] << "  " << dp[k] << endl;
19:    cout << "==========" << endl;
20:    ip = reinterpret_cast<int*>( realloc(ip, 7*sizeof(int)) );
21:    dp = reinterpret_cast<double*>(
22:                              realloc(dp, 7*sizeof(double)) );
23:    for ( int k(3); k <= 6; ++k )
24:    {
25:       ip[k] = k+1;
26:       dp[k] = 1.1*ip[k];
27:    }
28:    for ( int k(0); k <= 6; ++k )
29:       cout << ip[k] << "  " << dp[k] << endl;
30:    free( dp );
31:    free( ip );
32: } // main
```

Αποτέλεσμα:

```
1  1.1
2  2.2
3  3.3
==========
1  1.1
2  2.2
3  3.3
4  4.4
5  5.5
6  6.6
7  7.7
```

Αρχικώς, με τις *malloc()* και *calloc()*, πήραμε μνήμη για πίνακες με τρία στοιχεία (γρ. 10-11). Στη συνέχεια, με τη *realloc* (γρ. 20-22) επεκτείναμε τους πίνακες σε επτά στοιχεία για τον καθένα. Στο τέλος, με τη *free()* (γρ. 30-31), επιστρέφουμε τη μνήμη που πήραμε.

Πριν τελειώσουμε να πούμε ότι είναι *σαφώς προτιμότερη η χρήση των* `new` *και* `delete` και να επαναλάβουμε ότι, σε κάθε περίπτωση:

- ***Μνήμη που παίρνεις με* `new` *θα απελευθερώνεται με* `delete` *και μνήμη που παίρνεις με malloc(), calloc(), realloc() θα απελευθερώνεται με free().***

και ακόμη περισσότερο:

- ***Μη χρησιμοποιείς στο ίδιο πρόγραμμα τις συναρτήσεις δυναμικής μνήμης της C και τους* `new` *και* `delete`.**

Αυτό βέβαια δεν εύκολο όταν μέσα στο πρόγραμμά σου χρησιμοποιείς βιβλιοθήκες ή έτοιμα κομμάτια προγράμματος. Στην περίπτωση αυτή περιορίσου στην τήρηση του πρώτου κανόνα.

Τέλος, να πούμε κάτι που μπορεί να συμβαίνει και να κάνει τα παραπάνω να φαίνονται περίεργα: Σε ορισμένες υλοποιήσεις της C++ οι `new` και `delete` υλοποιούνται με τις *malloc()* και *free()*.

- Αν αναπτύξεις μια εφαρμογή σε ένα τέτοιο περιβάλλον μπορεί να μην έχεις πρόβλημα όσο και αν ανακατέψεις τους τελεστές της C++ με τις συναρτήσεις της C.
- Σε μια τέτοια εγκατάσταση μπορεί το πρόβλημα της κλήσης “`malloc(0)`” να μεταφερθεί και στον τελεστή `new`.

16.12 Για να Μη Ζηλεύουμε τη *realloc()*

Όπως είπαμε και παραπάνω, θα πρέπει να προτιμάς τους **new** και **delete** που είναι πιο βολικοί από τις συναρτήσεις. Βέβαια υπάρχει και η *realloc()*, που χρειάζεται αρκετά συχνά. Αργότερα θα μάθουμε ότι η C++ μας δίνει πολύ καλά εργαλεία[6], όπως το *vector*, το *map*, το *list*, που σου επιτρέπουν να έχεις δυναμικές δομές δεδομένων χωρίς να ανησυχείς για "**new**" και "**delete**".

Προς το παρόν όμως το πρόβλημα μπορεί να σου το λύσει το παρακάτω περίγραμμα συνάρτησης, που

- παίρνει το βέλος *p* προς έναν δυναμικό πίνακα με στοιχεία τύπου *T* και
- το αλλάζει σε βέλος προς νέον δυναμικό πίνακα με *nf* στοιχεία του ίδιου τύπου. Στα πρώτα *ni* στοιχεία του νέου πίνακα αντιγράφονται τα πρώτα *ni* στοιχεία του παλιού.

Η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(: :*domainError*)[7] αν κληθεί χωρίς να ισχύει η συνθήκη $0 \leq ni \leq nf$.

Η συνάρτηση θα δεχθεί βέλος 0 (μηδέν) αρκεί να μην έχει θετική τιμή η *ni*. Αλλιώς (*p* == 0 && *ni* > 0) θα ρίξει εξαίρεση *MyTmpltLibXptn*(: :*noArray*).

Φυσικά, η συνάρτηση θα ρίξει εξαίρεση *MyTmpltLibXptn*(: :*allocFailed*) αν δεν μπορέσει να πάρει την απαραίτητη μνήμη.

```
template< typename T >
void renew( T*& p, int ni, int nf )
{
   if ( ni < 0 || nf < ni )
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::domainError, ni, nf );
// 0 <= ni <= nf
   if ( p == 0 && ni > 0 )
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::noArray );
// (0 <= ni <= nf) && (p != 0 || ni == 0)
   try
   {
      T* temp( new T[nf] );
      for ( int k(0);  k < ni; ++k )  temp[k] = p[k];
      delete[] p;   p = temp;
   }
   catch( bad_alloc& )
   {
      throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed );
   }
} // void renew
```

Παρατηρήσεις: ▶

1. Οτιδήποτε και να πάει στραβά, αν ριχτεί εξαίρεση, δεν θα πειραχτεί ο αρχικός πίνακας.
2. Η *bad_alloc* μπορεί να προέλθει όχι μόνο από τη "**new T[nf]**" αλλά και από τις αντιγραφές "**temp[k] = p[k]**". Αυτό μπορεί να γίνει αν ο *T* έχει αντικείμενα που χρησιμοποιούν δυναμική μνήμη, όπως λέγαμε στην §16.8. Θα πρέπει να κάνουμε κάτι σαν αυτό που κάναμε στη *new2d*. Θα το μάθουμε αργότερα. ◀

Στη συνέχεια, ξαναδίνουμε το παράδειγμα που είδαμε πιο πάνω αλλά με τα εργαλεία της C++:

```
#include <iostream>
#include <new>

using namespace std;

template< typename T >
void renew( T*& p, int ni, int nf );
```

[6] Πρόκειται για περιγράμματα κλάσεων.

[7] Είδαμε τη *MyTmpltLibXptn* πιο πριν, στην §16.9.

```
int main()
{
   int*    ip( 0 );
   double* dp( 0 );

   ip = new int [3];  dp = new double [3];
   for ( int k = 0; k <= 2; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 2; ++k )
      cout << ip[k] << "   " << dp[k] << endl;
   cout << "==========" << endl;
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
   for ( int k = 3; k <= 6; ++k )
   {  ip[k] = k+1;  dp[k] = 1.1*ip[k];  }
   for ( int k = 0; k <= 6; ++k )
      cout << ip[k] << "   " << dp[k] << endl;
   delete [] dp;
   delete [] ip;
} // main
```

Η `renew(ip, 3, 7)` αλλάζει την τιμή του βέλους προς έναν δυναμικό πίνακα με 7 στοιχεία τύπου `int`. Στα 3 πρώτα στοιχεία αυτού του πίνακα αντιγράφονται τα στοιχεία του πίνακα που έδειχνε το *ip* πριν από την κλήση της συνάρτησης. Παρόμοια κάνει και η `renew(dp, 3, 7)` στον dp.

Η C++ ελαχιστοποιεί την ανάγκη για χρήση της *realloc* με το περίγραμμα κλάσης *vector* που υπάρχει στην πάγια βιβλιοθήκη περιγραμμάτων (Standard Template Library, STL).

16.13 Παραδείγματα

Θα δώσουμε τώρα τρία παραδείγματα χρήσης δυναμικής μνήμης που είναι τρεις παραλλαγές του δεύτερου προγράμματος (§15.14.2) της §15.14. Πριν προχωρήσουμε όμως θα πρέπει να τονίσουμε ότι *τα παραδείγματα δίνονται επειδή κρίνονται κατάλληλα για την επίδειξη χρήσης όσων είπαμε παραπάνω. Σε καμιά περίπτωση δεν εννοούμε ότι αυτές οι λύσεις είναι καλύτερες από την αρχική.*

Παράδειγμα 1 ☞

Αντί να διαβάζουμε τα δεδομένα του κάθε χημικού στοιχείου που μας ζητείται και να τα φυλάγουμε μετά τη συμπλήρωση/διόρθωση του στοιχείου αυτού

- *διαβάζουμε ολόκληρο το αρχείο σε έναν δυναμικό πίνακα,*
- *κάνουμε όλες τις ενημερώσεις στον πίνακα και*
- *φυλάγουμε στο αρχείο τον ενημερωμένο πίνακα.*

Αυτό το πρόγραμμα είναι απλούστερο από το αρχικό. Δηλώνουμε:

```
   GrElmn* grElmnTbl( 0 );
```

και μόλις μάθουμε τον μέγιστο ατομικό αριθμό –που είναι ίσος με το πλήθος των στοιχείων– ζητούμε την απαραίτητη μνήμη:

```
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
```

Για την περίπτωση αποτυχίας προσθέτουμε, μετά την ομάδα **try**, άλλη μια:

```
   catch ( bad_alloc& )
   {
      cout << "not enough memeory to load data" << endl;
   }
```

Στη συνέχεια φορτώνουμε το περιεχόμενο του αρχείου στον πίνακα με την:

```
      loadAllData( bInOut, grElmnTbl, maxAtNo );
```

όπου:

```
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekg( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_load( grElmnTbl[k], bInOut );
} // loadAllData
```

Η *editGrName* δεν μας βολεύει πια. Την τροποποιούμε:

```
void editGrNameMM( GrElmn& a )
{
   GrElmn_display( a, cout );
   string newGrName;
   cout << "new greek name: "; getline( cin, newGrName, '\n' );
   if ( !newGrName.empty() )
   {
      GrElmn_setGrName( a, newGrName );
   }
} // editGrNameMM
```

Όπως βλέπεις, δεν έχει πια στις παραμέτρους το ρεύμα απο/προς το αρχείο αφού η δουλειά γίνεται στην κύρια μνήμη (βάλαμε το "*MM*" για να μας θυμίζει τη «Main Memory»).

Την καλούμε, αφού πρώτα διαβάσουμε το ατομικό αριθμο, ως εξής:

```
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
```

(θυμίσου ότι το στοιχείο με ατομικό αριθμό *aa* βρίσκεται στη θέση *aa*-1 του πίνακα.)

Τελικώς, φυλάγουμε τον πίνακα με την:

```
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo )
{
   bInOut.seekp( 0 );
   for ( int k(0); k < maxAtNo; ++k )
      GrElmn_save( grElmnTbl[k], bInOut );
} // saveAllData
```

Οι συναρτήσεις *readRandom()* και *writeRandom()* δεν μας χρειάζονται. Οι *loadAllData()* και *saveAllData()* επεξεργάζονται το αρχείο ως σειριακό.

Το πρόγραμμά μας θα είναι κάπως έτσι:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
struct GrElmn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
// ΝΕΕΣ ΣΥΝΑΡΤΗΣΕΙΣ
void editGrNameMM( GrElmn& a );
void loadAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );
void saveAllData( fstream& bInOut,
                  GrElmn grElmnTbl[], int maxAtNo );

int main()
```

```
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );

   try
   {
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      grElmnTbl = new GrElmn[maxAtNo];
      loadAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            editGrNameMM( grElmnTbl[aa-1] );
         }
      } while ( aa != 0 );
      openFile( flNm, bInOut );
      saveAllData( bInOut, grElmnTbl, maxAtNo );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch ( bad_alloc& )
   {
      cout << "not enough memory to load data" << endl;
   }
   catch( ApplicXptn& x )
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Όπως βλέπεις:

- μια φορά παίρνουμε μνήμη (`grElmnTbl = new GrElmn[maxAtNo]`),
- μια φορά ανακυκλώνουμε (`delete[] grElmnTbl`).

Έτσι, δεν υπάρχει λόγος να πιάσουμε τη *bad_alloc* και να ρίξουμε δική μας.

❖❖❖

Παράδειγμα 2 ➢

Θα ξαναλύσουμε το προηγούμενο πρόβλημα με έναν άλλον τρόπο: *Θα κρατούμε σε πίνακα μόνο τις εγγραφές που ζητούνται από τον χρήστη και με το τέλος χρήσης του προγράμματος θα ενημερώνουμε το αρχείο.*

Η ουσιαστική διαφορά από την προηγούμενη περίπτωση είναι η εξής:

- Στο προηγούμενο πρόγραμμα με το άνοιγμα του αρχείου μπορούσαμε να υπολογίσουμε το μέγεθος του δυναμικού πίνακα.
- Τώρα το μέγεθος του δυναμικού πίνακα εξαρτάται από το πόσο θα δουλέψει ο χρήστης! Κάθε φορά που ο χρήστης αποφασίζει να διορθώσει ένα στοιχείο το μέγεθος του δυναμικού πίνακα θα αυξάνεται κατά 1.

Αυτό το τελευταίο μπορεί και να μην είναι έτσι ακριβώς. Ας πούμε, ότι ο χρήστης παίρνει το στοιχείο 74 (Tungsten), το μεταφράζει «Τουνγκστένιο» και μετά θυμάται ότι στα ελληνικά το λέμε «Βολφράμιο»! Τα δεδομένα για το στοιχείο αυτό βρίσκονται στον πίνακα

και δεν χρειάζεται να τα ξαναφορτώσει! Άρα μπορεί να κάνει και διορθώσεις χωρίς να βάζει και άλλα στοιχεία στον πίνακα.

Αυτό όμως βάζει μια άλλη απαίτηση: αναζήτηση στα στοιχεία του πίνακα με βάση τον ατομικό αριθμό. Αυτό μπορούμε να το αντιμετωπίσουμε: Θα πάρουμε τη *linSearch()* –όπως την κάναμε στην §12.2.1– και θα τη μετατρέψουμε σε περίγραμμα (§16.13.1). Αλλά, για τη χρησιμοποιήσουμε θα πρέπει να μπορούμε να κάνουμε εκχώρηση (τελεστής "=") και σύγκριση "!=". Θα πρέπει λοιπόν να επιφορτώσουμε τον τελεστή "!=" (και τον αντίθετό του "==") για τον *GrElmn*. Αυτό είναι πολύ απλό αν σκεφτούμε ότι αντικείμενα τύπου *GrElmn* είναι ίσα αν και μόνον αν έχουν ίδιο ατομικό αριθμό:

```
bool operator!=( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber != rhs.geANumber );  }

bool operator==( const GrElmn& lhs, const GrElmn& rhs )
{  return ( lhs.geANumber == rhs.geANumber );  }
```

Να σκεφτούμε τώρα πώς θα διαχειριστούμε τον δυναμικό πίνακα. Θα μπορούσαμε να καλούμε τη *renew()* κάθε φορά που ζητείται κάποιο στοιχείο που δεν υπάρχει στον πίνακα. Για σκέψου όμως τι σημαίνει αυτό:

- Όταν βάλουμε το δεύτερο στοιχείο θα κάνουμε μια αντιγραφή (του πρώτου).
- Όταν βάλουμε το τρίτο θα κάνουμε δύο αντιγραφές (του πρώτου και του δεύτερου).

κ.ο.κ. Έτσι, αν ο χρήστης, σε μια εκτέλεση του προγράμματος διορθώσει 12 διαφορετικά στοιχεία η *renew()* θα κάνει:

$$1 + 2 + 3 + \ldots + 11 = 66 \text{ αντιγραφές}$$

Η άλλη ακραία επιλογή είναι να κρατήσουμε χώρο για ολόκληρον τον πίνακα αλλά να χρησιμοποιήσουμε μόνο 12 από τις θέσεις του.

Ας εξετάσουμε μια ενδιάμεση περίπτωση: Να αυξάνουμε το μέγεθος τού πίνακα κατά πεντάδες. Έτσι, όταν πάρουμε τη δεύτερη πεντάδα θα έχουμε 5 αντιγραφές και όταν πάρουμε την τρίτη άλλες 10. Συνολικώς θα κάνουμε 15 αντιγραφές ενώ θα πάρουμε μόνον τρεις παραπάνω θέσεις.

Η λύση στο πρόβλημά μας είναι κάτι τέτοιο. Αλλά, θα αυξάνουμε κατά 5 ή κατά 10 ή κατά 80; Η απάντηση εξαρτάται από το συγκεκριμένο πρόβλημα κάθε φορά. Όπως είδες, με την υπόθεση εργασίας «*σε μια εκτέλεση του προγράμματος διορθώνει 12 διαφορετικά στοιχεία*», το «κβάντο» αύξησης 5 φαίνεται να δουλεύει μια χαρά. Αν λέγαμε ότι θα διορθωσει 35 με 40 στοιχεία ένα «κβάντο» 10 ή 15 θα ήταν πιο κατάλληλο.

Και πριν προχωρήσουμε παρακάτω να υπενθμίσουμε ότι η *linSearch* χρησιμοποιεί και φρουρό. Για να έχουμε, λοιπόν, *n* στοιχεία αποθηκευμένα και να κάνουμε και αναζητήσεις χρειαζόμαστε πίνακα με *n*+1 θέσεις.

Για να ανταποκριθούμε σε όλες τις απαιτήσεις που βάλαμε πιο πάνω θα πρέπει να δηλώσουμε για τον δυναμικό πίνακα τα εξής:

```
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων
```

και όταν πάρουμε για πρώτη φορά μνήμη:

```
      grElmnTbl = new GrElmn[incr];
      nReserved = incr;
'     nElmn = 0;
```

Κάθε φορά που θα θέλουμε να εισαγάγουμε ένα νέο στοιχείο –με ατομικό αριθμό *aa*– στον πίνακα θα δουλεύουμε ως εξής:

```
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
```

```
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
```

Πρόσεξε τα εξής σημεία:

- Αυξάνουμε το μέγεθος του πίνακα όταν *nElmn*+1 == *nReserved*. Το «+1» μας επιτρέπει να έχουμε μια διαθέσιμη θέση στο τέλος για να βάζει η *linSearch* τον φρουρό.
- Καλούμε τη *renew()* με την `renew(grElmnTbl, nElmn, nReserved+incr)`. Το νέο μέγεθος του πίνακα θα είναι `nReserved+incr`. Από τον αρχικό πίνακα θα αντιγραφούν τα `nElmn` στοιχεία.
- Το νέο στοιχείο διαβάζεται από το αρχείο στη θέση `nElmn` του πίνακα. Μετά την αύξηση της τιμής της `nElmn` η θέση θα είναι `nElmn-1`.

Όλα αυτά τα κρύβουμε μέσα σε μια συνάρτηση που σιγουρεύει ότι τα δεδομένα του στοιχείου που μας ενδιαφέρει βρίσκονται στον πίνακα:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos )
{
   GrElmn oneElmn( aa );
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = nElmn - 1;
   }
   else
      pos = lPos;
} // elmntInTable
```

Να δούμε τι γίνεται με αυτήν τη συνάρτηση:

- Κατ' αρχάς έχει πολλές (7) παραμέτρους. Οι τέσσερις από αυτές έχουν να κάνουν με τον δυναμικό πίνακα και οι τρεις από αυτές είναι αναφοράς. Πράγματι:
  - Είναι πολύ πιθανό να κάνει εισαγωγή νέου στοιχείου στον πίνακα οπότε μπορεί να αλλάξει το πλήθος στοιχείων *nElmn*.
  - Μπορεί να χρειαστεί να μεγαλώσει ο πίνακας –με κλήση της *renew*– οπότε αλλάζει το βέλος *grElmnTbl* και το πλήθος των δεσμεμένων στοιχείων *nReserved*.
- Πριν από οτιδήποτε άλλο, καλείται η *linSearch()* για να μας φέρει στην *lPos* τη θέση του στοιχείου, αν υπάρχει στον πίνακα. Η *linSearch()* περιμένει ότι η τιμή που θα αναζητήσει θα είναι του ίδιου τύπου με τα στοιχεία του πίνακα. Για αυτόν ακριβώς τον λόγο δηλώνουμε την *oneElmn*.
- Αφού η σύγκριση "`!=`" γίνεται με χρήση μόνο του ατομικού αριθμού, για να κάνουμε τη δουλειά μας αρκεί να βάλουμε *oneElmn*.*geANumber* τη τιμή της παραμέτρου *aa*, Αυτό μπορούμε να το κάνουμε:
  - Με την εντολή "`oneElmn.geANumber = aa`".
  - Με τη δήλωση της *oneElmn* αν έχουμε τον κατάλληλο δημιουργό. Προτιμήσαμε αυτή τη λύση και γράψαμε τον

```
struct GrElmn
{
// . . .
   GrElmn( int aAN=0 ) {  geANumber = aAN;  }
}; // GrElmn
```

- ◦ Αν έχουμε αυτόν τον δημιουργό δεν χρειάζεται να δηλώσουμε τη μεταβλητή *oneElmn* αλλά μπορούμε να καλέσουμε τη *linSearch()* ως εξής:

```
   int lPos( linSearch(grElmnTbl, nElmn, 0, nElmn-1, GrElmn(aa)));
```

- Είδαμε παραπάνω τι κάνουμε αν το στοιχείο που ζητάει ο χρήστης δεν υπάρχει στον πίνακα (*lPos* < 0). Στην περίπτωση αυτή τα δεδομένα φορτώνονται από το αρχείο στην τελευταία χρησιμοποιούμενη θέση του πίνακα και επιστρέφουμε ως τιμή της *pos* το `nElmn - 1`.
- Αν τα δεδομένα υπάρχουν στη θέση *lPos* αυτή επιστρέφεται ως τιμή της *pos*.

Όταν ο χρήστης τελειώσει τη δουλειά του το περιεχόμενο του πίνακα φυλάγεται στο αρχείο με την:

```
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
```

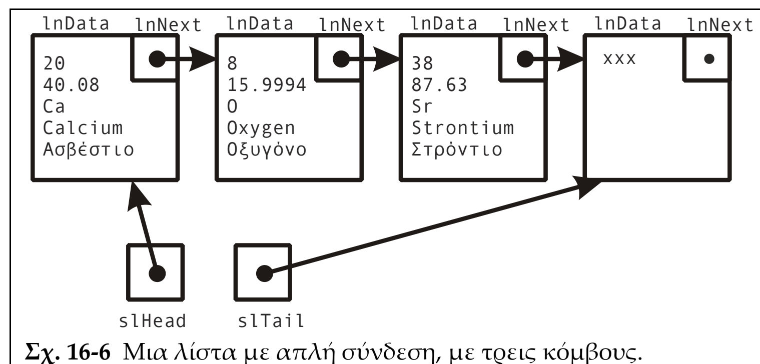
όπου:

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn )
{
   for ( int k(0); k < nElmn; ++k )
      writeRandom( grElmnTbl[k], bInOut );
} // saveUpdateTable
```

Τώρα το πρόγραμμα θα είναι ως εξής:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

template< typename T >
void renew( T*& p, int ni, int nf );
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x );

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
   GrElmn( int aAN=0 ) {  geANumber = aAN;   }
}; // GrElmn

// Συναρτήσεις GrElmn_... ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
bool operator!=( const GrElmn& lhs, const GrElmn& rhs );
bool operator==( const GrElmn& lhs, const GrElmn& rhs );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
```

```
void saveUpdateTable( fstream& bInOut,
                      const GrElmn grElmnTbl[], int nElmn );
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, unsigned int& pos );

int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );
   GrElmn* grElmnTbl( 0 );
   const unsigned int incr( 5 ); // κβάντο αύξησης
   unsigned int nElmn;           // αριθμός στοιχείων σε χρήση
   unsigned int nReserved;       // αριθμός δεσμευμένων στοιχείων

   try
   {
      try {  grElmnTbl = new GrElmn[incr];  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "main", ApplicXptn::allocFailed );  }
      nReserved = incr;
      nElmn = 0;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            unsigned int pos;
            elmntInTable( grElmnTbl, nElmn, nReserved, incr,
                          bInOut, aa, pos );
            editGrNameMM( grElmnTbl[pos] );
         }
      } while ( aa != 0 );
      saveUpdateTable( bInOut, grElmnTbl, nElmn );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      delete[] grElmnTbl;
   }
   catch( ApplicXptn& x )
   {
      switch ( x.errCode )
      {
      // ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ
         case ApplicXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected ApplicXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( ApplicXptn
   catch( MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
         case MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
```



Σχ. 16-6 Μια λίστα με απλή σύνδεση, με τρεις κόμβους.

```
            break;
         case MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στο πρόγραμμα αυτό παίρνουμε δυναμική μνήμη στη **main** (μια φορά) και στη *renew()*. Παρ' όλο που δεν υπάρχει περίπτωση σύγχυσης, αφού από τη *renew()* ρίχνονται εξαιρέσεις τύπου *MyTmpltLibXptn*, προτιμήσαμε να πιάνουμε και στη **main** τη *bad_alloc* και να ρίχνουμε μια δική μας *ApplicXptn(::allocFailed)*.

Στη διαχείριση των εξαιρέσεων *ApplicXptn* προσθέσαμε μια επι πλέον περίπτωση *ApplicXptn::allocFailed*.

Παράδειγμα 3

Τώρα θα ξαναγράψουμε το πρόγραμμα του Παραδ. 2 με την εξής διαφορά: *Θα κρατούμε τις εγγραφές που ζητάει ο χρήστης όχι σε δυναμικό πίνακα αλλά σε μια λίστα με (απλή) σύνδεση.*

Η **λίστα με απλή σύνδεση** (simply linked list) –μια υλοποίηση της *ακολουθίας* (sequence)– είναι μια ευρύτατα χρησιμοποιούμενη δομή δεδομένων. Θα δούμε εδώ ένα τμήμα υλοποίησης μιας τέτοιας λίστας με λογική στοίβας.

Η παράσταση που θα υλοποιήσουμε φαίνεται στο Σχ. 16-6. Όπως βλέπεις, η λίστα είναι μια ακολουθία από *κόμβους* (nodes). Ο «κόμβος-φρουρός» στο τέλος απλουστεύει ορισμένους αλγόριθμους διαχείρισης της λίστας.

Κάθε κόμβος έχει δεδομένα (στην περίπτωσή μας τύπου *GrElmn*) και ένα βέλος-σύνδεσμο προς τον επόμενο κόμβο. Μπορούμε να τον υλοποιήσουμε στο πρόγραμμά μας με αντικείμενα τύπου:

```
struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode
```

Ένα βέλος δείχνει την αρχή της λίστας. Η ύπαρξη και δεύτερου βέλους που δείχνει τον φρουρό στο τέλος της λίστας απλουστεύει –όπως και ο ίδιος ο φρουρός, ορισμένους αλγόριθμους. Μπορούμε να πούμε λοιπόν ότι θα έχουμε:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

Στο Σχ. 16-7 βλέπεις μια κενή λίστα. Από αυτήν την εικόνα μπορούμε εύκολα να βγάλουμε τον ερήμην δημιουργό:

```
struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // Slist
```

Εδώ βλέπουμε ξανά να ρίχνεται εξαίρεση από δημιουργό. Βλέπουμε όμως και κάτι ακόμη: πώς παιρνουμε μνήμη για έναν κόμβο της λίστας. Με τον ίδιο τρόπο θα παίρνουμε και για τους υπόλοιπους κόμβους. Αλλά τι θα γίνει όταν θελήσουμε να ανακυκλώσουμε αυτή τη μνήμη; Και *η ανακύκλωση θα γίνει κομβο προς κόμβο·* δεν υπάρχει "`delete`" που να καθαρίζει όλη τη λίστα.

Να δούμε τώρα τι θέλουμε να κάνουμε με μια τέτοια λίστα:

- Εισαγωγή δεδομένων ενός στοιχείου στη λίστα (σε νέο κόμβο).
- Αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου.
- Φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.
- Ανακύκλωση όλων των κόμβων της λίστας.

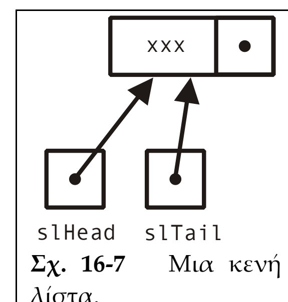
Ξεκινούμε με τη συνάρτηση εισαγωγής δεδομένων στη λίστα που δεν είναι και τόσο μπερδεμένη:[8]

```
void SList_push_front( SList& lst, const GrElmn& aData )
{
   ListNode* pnln( 0 );
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw ApplicXptn( "SList_push_front",
                        ApplicXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = lst.slHead;
   lst.slHead = pnln;
} // SList_push_front
```

Ας δούμε τη λειτουργία της με ένα παράδειγμα από το «κτίσιμο» της λίστας του Σχ. 16-6: Έχουμε μια λίστα στην οποίαν υπάρχει μόνο το Στρόντιο και θέλουμε να εισαγάγουμε και το Οξυγόνο. Τα δεδομένα του Οξυγόνου υπάρχουν στην παράμετρο *aData*.

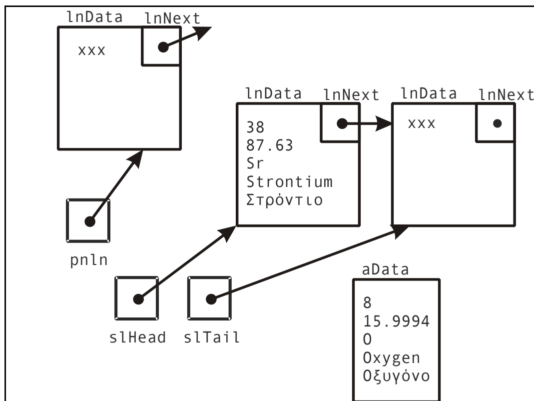
Αν προσπαθήσουμε να πάρουμε δυναμική μνήμη κάπως απρόσεκτα με μια "`lst.slHead = new ListNode`" *χάνουμε την αρχή της λίστας*. Επιλέγουμε λοιπόν να κάνουμε το εξής:

- Παίρνουμε μνήμη χρησιμοποιώντας ένα βοηθητικό βέλος, το *pnln* με την "`pnln = new ListNode`". Στο Σχ. 16-8α βλέπεις την

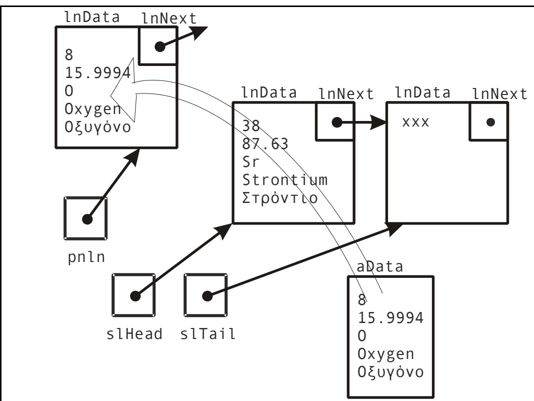


Σχ. 16-7 Μια κενή λίστα.

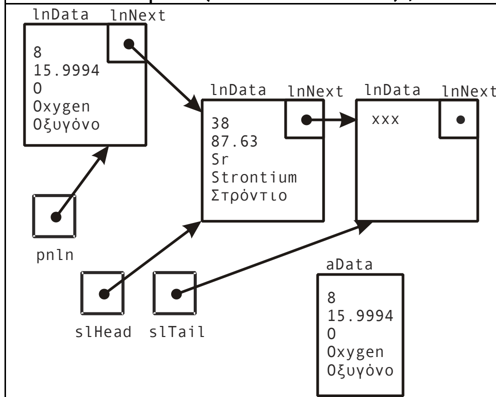
[8] Το όνομα από την STL.



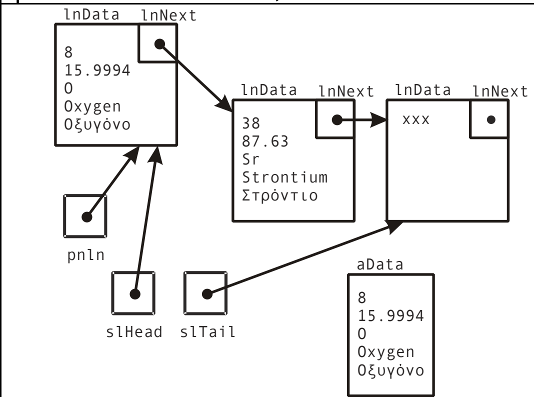
Σχ. 16-8α Μετά την εκτέλεση της εντολής:
`ListNode* pnln( new ListNode );`



Σχ. 16-8β Μετά την εκτέλεση της εντολής:
`pnln->lnData = aData;`



Σχ. 16-8γ Μετά την εκτέλεση της εντολής:
`pnln->lnNext = lst.slHead;`



Σχ. 16-8δ Μετά την εκτέλεση της εντολής:
`lst.slHead = pnln;`

κατάσταση που διαμορφώνεται. Η λίστα δεν έχει πειραχτεί ακόμη.

- Στον νέο κόμβο, στο μέλος *lnData* αντιγράφουμε τα δεδομένα που θέλουμε να εισαγάγουμε στη λίστα με την "`pnln->lnData = aData`". Η λίστα δεν έχει πειραχτεί ακόμη. Η κατάσταση φαίνεται στο Σχ. 16-8β.
- Με την "`pnln->lnNext = lst.slHead`" συμπληρώνεται ο νέος κόμβος. Το βέλος *lnNext* δείχνει τον μέχρι τώρα πρώτο κόμβο της λίστας. Όπως βλέπεις στο Σχ. 16-8γ έχουμε μια «νέα είσοδο» στη λίστα που κατά τα άλλα δεν έχει πειραχτεί.
- Με την "`lst.slHead = pnln`" ο νέος κόμβος ενσωματώνεται στη λίστα (Σχ. 16-8δ): Ξεκινώντας από την *slHead* περνάς από τον νέο κόμβο και από αυτόν πηγαίνεις στον προηγούμενο πρώτο (με το Στρόντιο).

Με το τέλος εκτέλεσης της συνάρτησης το βέλος *pnln* παύει να υπάρχει.

Σε δύο περιπτώσεις, μετά από αντιγραφές βελών, είχαμε δύο βέλη να δείχνουν τον ίδιο στόχο: στα Σχ. 16-8γ και 16-8δ, αλλά δεν υπήρχε οποιοδήποτε πρόβλημα.

Ο τρόπος αυτός δεν είναι ο μοναδικός· να και ένας άλλος το ίδιο σωστός:

```
   ListNode* pnln( lst.slHead );
   lst.slHead = new ListNode;
   (lst.slHead)->lnData = aData;
   (lst.slHead)->lnNext = pnln;
```

Η «*αναζήτηση κόμβου που έχει τα δεδομένα ενός στοιχείου*» μπορεί να γίνει στη λίστα όπως περίπου γίνεται στον πίνακα με τη *linSerch*:

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   ListNode* p( lst.slHead );
   while ( p->lnData != aData && p->lnNext != lst.slTail )
      p = p->lnNext;
   return (p->lnData == aData ? p : 0 );
```

```
} // SList_listSearch
```

Το βέλος *p* ξεκινάει δείχνοντας το στοιχείο που δείχνει και το *lst.slHead* και προχωρεί από στοιχείο σε στοιχείο με την "`p = p->lnNext`". Διασχίζει τη λίστα κόμβο προς κόμβο με την "`p = p->lnNext`" μέχρι

- Να βρει τα στοιχεία, αν υπάρχουν ("`p->lnData != aData`") ή
- Να φτάσει στον φρουρό ("`p->lnNext != lst.slTail`").

Αν έχει βρει την τιμή ("`p->lnData == aData`") η τιμή που επιστρέφει η συνάρτηση είναι *p*· αλλιώς επιστρέφει `0`.

Αν στον κόμβο-φρουρό αντιγράψουμε την τιμή που ψάχνουμε δεν θα απαλλαγούμε από τη μια σύγκριση της **while**; Βεβαίως, αυτή η τεχνική είναι η αντίστοιχη αυτής που μάθαμε στη γραμμική αναζήτηση σε πίνακα. Φυσικά, αυτό απαγορεύεται λόγω του "`const SList& lst`" αλλά ξέρουμε πώς θα αντιμετωπίσουμε: με τυποθεώρηση **const**.

```
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData )
{
   SList& ncLst( const_cast<SList&>(lst) );
   ncLst.slTail->lnData = aData;
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
   return (p != lst.slTail ? p : 0 );
} // SList_listSearch
```

Η *ncLst* είναι η *lst* αλλά τύπου *SList*& –χωρίς "`const`"– και μπορούμε να δώσουμε:

```
   ncLst.slTail->lnData = aData;
```

Έτσι, η **while** γίνεται:

```
   ListNode* p( lst.slHead );
   while ( p->lnData != aData )  p = p->lnNext;
```

και η **return**:

```
   return (p != lst.slTail ? p : 0 );
```

(επίστρεψε το *p* αν δεν δείχνει τον φρουρό, αλλιώς το 0)

Η *linSearch()* επιστρέφει δείκτη στοιχείου πίνακα. Η *SList_listSearch* επιστρέφει βέλος προς κόμβο που έχει μέλος τύπου *GrElmn*. Έτσι, τώρα θα καλέσουμε την *editGrNameMM* αλλά κάπως διαφορετικά:

```
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
```

Εδώ είδαμε και πώς διασχίζουμε ολόκληρη τη λίστα: «*Το βέλος p ξεκινάει δείχνοντας το στοιχείο που δείχνει και το lst.slHead και προχωρεί από στοιχείο σε στοιχείο με την* "`p = p->lnNext`" *... μέχρι ... να φτάσει στον φρουρό* ("`p->lnNext != lst.slTail`").»

Πώς θα κάνουμε λοιπόν τη «*φύλαξη των περιεχόμενων των κόμβων της λίστας στο αρχείο.*» Τη διασχίζουμε και φυλάγουμε το περιεχόμενο του κάθε κόμβου:

```
void saveUpdateList( fstream& bInOut, const SList& lst )
{
   for ( ListNode* p(lst.slHead); p != lst.slTail; p = p->lnNext )
      writeRandom( p->lnData, bInOut );
} // saveUpdateList
```

Παρομοίως γίνεται και η «*ανακύκλωση όλων των κόμβων της λίστας.*»

```
void SList_deleteAll( SList& lst )
{
   while ( lst.slHead != lst.slTail )
   {
      ListNode* p( lst.slHead );
      lst.slHead = (lst.slHead)->lnNext;
      delete p;
   }
   delete lst.slHead;
```

```
   lst.slTail = lst.slHead = 0;
} // SList_deleteAll
```

Πρόσεξε πώς γίνεται η διαγραφή του (εκάστοτε) πρώτου κόμβου:

- Φυλάγουμε στο *p* την τιμή του *lst.slHead* που δείχνει τον πρώτο κόμβο.
- Προχωρούμε το *lst.slHead* στον επόμενο κόμβο.
- Ανακυκλώνουμε αυτόν που δείχνει το *p*.

Αυτή η διαδικασία τελειώνει όταν "`lst.slHead == lst.slTail`": και τα δύο βέλη δείχνουν τον φρουρό. Η "`delete lst.slHead`" ανακυκλώνει και τον φρουρό.

Σε τι κατάσταση βρίσκεται η λίστα τώρα; Είναι άδεια; Όχι! *Δεν υπάρχει λίστα!* Δηλαδή δεν μπορείς να ξανακάνεις εισαγωγή στοιχείου με *SList_push_front()*. Αυτή η κατάσταση είναι ανιχνεύσιμη μετά την εκτέλεση των εκχωρήσεων:

```
   lst.slTail = lst.slHead = 0;
```

Να δούμε τώρα το πρόγραμμα:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

struct GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

// Συναρτήσεις GrElmn
// ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ

struct ListNode
{
   GrElmn    lnData;
   ListNode* lnNext;
}; // ListNode

struct SList
{
   ListNode* slHead;
   ListNode* slTail;
   SList()
   {
      try {  slHead = new ListNode;  }
      catch( bad_alloc& )
      {  throw ApplicXptn( "SList", ApplicXptn::allocFailed );  }
      slHead->lnNext = 0;  slTail = slHead;
   } // SList
}; // SList

void SList_push_front( SList& lst, const GrElmn& aData );
ListNode* SList_listSearch( const SList& lst,
                            const GrElmn& aData );
void SList_deleteAll( SList& lst );

void openFile( string& flNm, fstream& bInOut );
void countRecords( fstream& bInOut, unsigned int& noOfRecords );
void readAtNo( int maxAtNo, int& aa );
void readRandom( GrElmn& a, istream& bin, int atNo );
void writeRandom( const GrElmn& a, ostream& bout );
void editGrNameMM( GrElmn& a );
void saveUpdateList( fstream& bInOut, const SList& lst );
void elmntInList( SList& lst,
                  fstream& bInOut, int aa, ListNode*& pos );
```

```
int main()
{
   fstream bInOut;
   string  flNm( "elementsGr.dta" );

   try
   {
      SList   lst;
      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            ListNode* pos;
            elmntInList( lst, bInOut, aa, pos );
            editGrNameMM( pos->lnData );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
      SList_deleteAll( lst );
   }
   catch( ApplicXptn& x )
   // ΟΠΩΣ ΣΤΟ ΠΡΟΗΓΟΥΜΕΝΟ
   catch( ... )
   {
      cout << "unexpected exception" << endl;
   }
} // main
```

Στις λίστες θα ξαναγυρίσουμε.

Πρόσεξε τώρα ένα σημαντικό πρόβλημα που έχουν αυτές οι τρεις λύσεις: *η καθυστέρηση που υπάρχει από τη στιγμή που κάνεις την ενημέρωση μέχρι να περάσει στο αρχείο.* Όταν λέμε «*καθυστέρηση*» την εννοούμε με όρους ανθρώπου και όχι υπολογιστή, π.χ. συμπληρώνω πέντε στοιχεία πάω για καφέ και τσιγάρο και επιστρέφω για τα υπόλοιπα. Αν λοιπόν συμβεί κάποιο είδος «system crash» χάνεις όλη τη δουλειά που έχεις κάνει εν τω μεταξύ (ενώ με το αρχικό πρόγραμμα το πολύ που μπορείς να χάσεις είναι η τελευταία εγγραφή).

Όμως, εκτός από αυτήν την περίπτωση, πρόσεξε ότι η *GrElmn_save()* μπορεί να ρίξει εξαίρεση *ApplicXptn*(::*cannotWrite*). Πώς μπορεί να γίνει αυτό; Για παράδειγμα, έχεις το αρχείο σε memory stick και κατά λάθος βγήκε από τη θέση του! Παρόμοια ζημιά μπορεί να γίνει –στα Παραδ. 2 και 3– και στην περίπτωση που η *GrElmn_load()* ρίξει εξαίρεση *ApplicXptn* (::*cannotRead*). Όπως είναι γραμμένα τα προγράμματα δεν υπάρχει πρόβλεψη για τέτοια καταστροφή. Για σκέψου όμως, κάτι θα μπορέσεις να σκαρφιστείς...

Ειρήσθω εν παρόδω ότι στις περιπτώσεις αυτές έχουμε και «διαρροή μνήμης» αλλά, αν έχεις χάσει όλη τη δουλειά που έκανες, μάλλον δεν σε ενδιαφέρει και τόσο.

16.13.1 Το Περίγραμμα *linSearch()*

Μετατρέπουμε σε περίγραμμα τη *linSearch* όπως την είδαμε στην §12.2.1

```
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
{
   if ( v == 0 && n > 0 )
      throw MyTmpltLibXptn( "linSearch",
                            MyTmpltLibXptn::noArray );
   int fv( -1 );
   if ( v != 0 && (0 <= from && from <= upto && upto < n) )
   {
      T* ncv( const_cast<T*>(v) );
      T  save( v[upto+1] );  // φύλαξε το v[upto+1]
      ncv[upto+1] = x;       // φρουρός
      int k( from );
      while ( v[k] != x )  ++k;
      if ( k <= upto )  fv = k;
               else  fv = -1;
      ncv[upto+1] = save;    // όπως ήταν στην αρχή
   // (from <= fv <= upto && v[fv] == x) ||
   //               (fv == -1 && (∀j:from..upto • v[j] != x))
   }
   return fv;
} // linSearch
```

Πέρα από τις αλλαγές από "`int`" σε "*T*" πρόσεξε τα εξής σημεία:

- Ελέγχουμε μήπως "`v == 0`". Στην περίπτωση αυτήν, αν το πλήθος στοιχείων του πίνακα *n* είναι 0 επιστρέφουμε τιμή "`-1`" ("δεν το βρήκα") αλλιώς ρίχνουμε εξαίρεση.
- Για να μπορεί να γίνει εξειδίκευση στον τύπο *T*, θα πρέπει να έχουμε για τον τύπο αυτόν:
  - Τελεστή εκχώρησης που να δουλεύει σωστά (αν δεν καταλαβαίνεις τι θα πει αυτό διάβασε, για παράδειγμα, αυτά που λέμε στην §16.8) για τις εκχωρήσεις "`ncv[upto+1] = x`" και "`ncv[upto+1] = save`". Στην περίπτωσή μας αυτός που υπάρχει αυτομάτως δουλεύει σωστά.
  - Τελεστή σύγκρισης "`!=`" για τη σύγκριση "`v[k] != x`" στη **while**.
  - *Δημιουργό αντιγραφής* που να δουλεύει σωστά! Τι είναι πάλι αυτό; Θα το μάθουμε αργότερα. Απλώς να πούμε ότι αυτός είναι που εκτελείται για τη δήλωση "`T save(v[upto+1])`".[9] Στην περίπτωσή μας και αυτός, όπως ο τελεστής εκχώρησης, υπάρχει αυτομάτως και δουλεύει σωστά.

Πάντως θα πρέπει να επιστήσουμε την προσοχή σου στις πολλές αντιγραφές που –αν τα αντικείμενα του τύπου *T* είναι μεγάλα– μπορεί να καθυστερούν την εκτέλεση του προγράμματός σου. Και να σκεφτείς ότι για τη σύγκριση χρειαζόμαστε μόνο το κλειδί, που συνήθως είναι πολύ πιο μικρό...

16.13.2 Χωρίς τη *linSearch()*

Η *linSearch* είναι ένα καλό εργαλείο από εκπαιδευτική άποψη, αλλά η C++ μας προσφέρει ένα περίγραμμα συνάρτησης –με το όνομα (`std::`)*find*– που κάνει την ίδια δουλειά: γραμμική αναζήτηση στα στοιχεία ενός πίνακα (και όχι μόνο). Θα το περιγράψουμε τώρα με τους περιορισμούς που υπάρχουν από αυτά που ξέρουμε μέχρι τώρα. Αργότερα, θα τη δούμε πληρέστερα.

Μπορείς, προς το παρόν, να σκέφτεσαι το περίγραμμα ως εξής:

```
template< typename T >
```

---

[9] Αν δεν θέλεις να ανησυχείς για δημιουργούς αντιγραφής και άλλα παρόμοια γράψε:

```
      T  save;
      save = v[upto+1];  // φύλαξε το v[upto+1]
```

```
T* find( T* first, T* last, T value )
```

όπου:

- Ο τύπος *T* έχει ορισμένη τη σύγκριση για ισότητα ("==").
- Η `[` *first*`..`*last* `)` είναι μια περιοχή βελών που μπορούμε να διασχίσουμε –ξεκινώντας από τη *first*– με μια μεταβλητή-βέλος `T* p` με την πράξη "`++p`". *Η last είναι η πρώτη θέση μετά την περιοχή αναζήτησης και δεν περιλαμβάνεται σε αυτήν.*
- *value* είναι η τιμή που αναζητούμε.

Η *find()* θα επιστρέψει βέλος προς την πρώτη θέση *fPos* για την οποίαν θα ισχύει η `*fPos == value`. Αν δεν βρει τη *value* επιστρέφει *last*.

Αν λοιπόν έχουμε έναν πίνακα ar –συμβατικό ή δυναμικό– με στοιχεία τύπου *T* για να ψάξουμε με τη *linSearch()* θα πρέπει να δώσουμε:

```
   unsigned int size, from, upto;
   int          ndx;
   T            value;
// . . .
   ndx = linSearch( ar, size, from, upto, value );
   if ( ndx >= 0 ) // βρέθηκε
   // . . .
```

ενώ για να ψάξουμε με τη *find* θα πρέπει να δώσουμε:

```
   unsigned int from, upto,;
   T*           pos;
   T            value;
// . . .
   pos = find( ar+from, ar+upto, value );//10
   if ( pos != ar+upto ) // βρέθηκε
   // . . .
```

Γυρνώντας στο Παράδ. 2, θα μπορούσαμε να γράψουμε τη *elmntInTable* ως εξής:

```
void elmntInTable( GrElmn*& grElmnTbl, unsigned int& nElmn,
                   unsigned int& nReserved, unsigned int incr,
                   fstream& bInOut, int aa, GrElmn*& pos )
{
   GrElmn oneElmn( aa );
   GrElmn* lPos( find(grElmnTbl, grElmnTbl+nElmn, GrElmn(aa)) );
   if ( lPos == grElmnTbl+nElmn ) // αν δεν υπάρχει
   {
      if ( nElmn+1 == nReserved )
      {
         renew( grElmnTbl, nElmn, nReserved+incr );
         nReserved += incr;
      }
      readRandom( grElmnTbl[nElmn], bInOut, aa );
      ++nElmn;
      pos = &grElmnTbl[nElmn-1];
   }
   else
      pos = lPos;
} // elmntInTable
```

Να επισημάνουμε τις διαφορές:

- Η τελευταία παράμετρος είναι τώρα: `GrElmn*& pos`.
- Παρομοίως, η *lPos* είναι τύπου *GrElmn**.
- Πρόσεξε πώς καλούμε τη *find* για να αναζητήσει τη `GrElmn(aa)` σε ολόκληρον τον πίνακα *grElmnTbl*:
  - Πρώτη παράμετρος ο πίνακας (η αρχή του),

[10] Ή, αν προτιμάς:

```
   pos = find( &ar[from], &ar[upto], value );
```

- Δεύτερη παράμετρος μια θέση μετά το τελευταίο στοιχείο, Να υπενθυμίσουμε ότι: `grElmnTbl+nElmn` σημαίνει `&grElmnTbl[nElmn]`.

Μετά την κλήση ελέγχουμε το *αν δεν βρέθηκε η τιμή* ελέγχοντας τη συνθήκη *lPos* == *grElmnTbl*+*nElmn*.

Μπορούμε να χρησιμοποιήσουμε τη *find* για αναζητήσεις στη λίστα; Όχι για δύο λόγους:

- Δεν διασχίζουμε τη λίστα με την "`++p`" αλλά με την "`p = p->lnNext`".
- Αυτό που αναζητούμε δεν είναι ολόκληρος ο κόμβος αλλά μέλος του κόμβου.

Αργότερα θα γνωρίσουμε τα κατάλληλα εργαλεία για να το καταφέρουμε και αυτό.

16.13.3 "*reserved* + *incr*" ή "2**reserved*"

Ο τρόπος διαχείρισης της δυναμικής μνήμης δεν είναι ο καλύτερος. Ας πούμε ότι ξεκινούμε με *reserved* == *incr* στοιχεία στον πίνακα και τελικώς εισάγουμε *n* τιμές. Θα πάρουμε μνήμη *n*/*incr* φορές.[11] Την *k*-οστή φορά θα πάρουμε μνήμη για *k***incr* στοιχεία και θα κάνουμε (*k*-1)**incr* αντιγραφές.

Συνολικώς, το πλήθος των αντιγραφών θα είναι:

$$0*incr + 1*incr + 2*incr + \ldots + ((n/incr)-1)*incr = (n/incr)*((n/incr)-1)*incr/2 = O(n^2)$$

Στη βιβλιοθήκη της C++ (STL), κάθε φορά που χρειάζεται δυναμική μνήμη, παίρνει διπλάσια από αυτήν που ήδη έχει. Έτσι, αν ξεκινήσει με μνήμη για *incr* στοιχεία, την *k*-οστή φορά θα πάρει μνήμη για $2^{k-1}*incr$ στοιχεία και θα κάνει $2^{k-2}*incr$ αντιγραφές ($k \geq 2$: την πρώτη φορά δεν γίνονται αντιγραφές). Την τελευταία φορά θα έχουμε πάρει μνήμη για $n == 2^{p-1}*incr$ στοιχεία. Από αυτήν έχουμε: $p = log_2(n/incr)+1$.

Οι αντιγραφές που θα γίνουν:

$$2^0*incr + 2^1*incr + 2^2*incr + \ldots + 2^{p-2}*incr = incr*(2^{p-1}-1) = O(n)$$

Με αυτόν τον τρόπο δουλεύουν οι *παραχωρητές μνήμης* (memory allocators) των *περιεχόντων* (containers) της βιβλιοθήκης της C++.

Εμείς θα χρησιμοποιούμε τον «αργό τρόπο» μέχρι να γυρίσουμε στην STL, μια και όταν χρησιμοποιούμε δυναμική παραχώρηση μνήμης άλλα είναι αυτά που θα θέλουμε να δείξουμε. Πάντως, εσύ, αν θέλεις, μπορείς να χρησιμοποιείς τον «γρήγορο τρόπο». Πιο πάνω, στο Παράδ. 2, είδαμε τις εντολές:

```
renew( grElmnTbl, nElmn, nReserved+incr );
nReserved += incr;
```

Μπορείς να τις αλλάξεις σε:

```
renew( grElmnTbl, nElmn, 2*nReserved );
nReserved *= 2;
```

16.14 Προβλήματα Ασφάλειας

Στην παράγραφο αυτή θα ασχοληθούμε με δυο συστάσεις του (CERT 2009):

- ***Καθάρισε ευαίσθητες πληροφορίες αποθηκευμένες σε ανακυκλώσιμους πόρους που επιστρέφονται για να ξαναχρησιμοποιηθούν.***[12]

και

- ***Εξασφάλισε ότι δεν γράφονται στον δίσκο ευαίσθητα δεδομένα.***[13]

---

[11] Για να μη μπερδεύεσαι με ατελείς διαιρέσεις θεώρησε ότι $n = 2^N*incr$.

[12] Σύσταση *MEM*03: "Clear sensitive information stored in reusable resources returned for reuse."

[13] Σύσταση *MEM*06: "Ensure that sensitive data is not written out to disk."

Θα ξεκινήσουμε ξαναδίνοντας ένα παράδειγμα που είδαμε στην §16.3. Με τη βοήθεια δύο βελών:

```
   double* a1( new double );
   double* a2;
```

είχαμε υλοποιήσει δύο δυναμικές μεταβλητές **a*1, **a*2. Αφού τους δώσαμε τιμές, η

```
   cout << *a1 << "   " << *a2 << endl;
```

μας έδωσε:

```
1.23  4.56
```

Στη συνέχεια τις ανακυκλώσαμε:

```
   delete a1;  delete a2;
```

ξαναϋλοποιήσαμε την **a*1 και είδαμε την τιμή της πριν προσπαθήσουμε να την ορίσουμε:

```
   a1 = new double;
   cout << *a1 << endl;
```

Αποτέλεσμα:

```
4.56
```

Πώς έγινε αυτό; Η **a1* υλοποιήθηκε τη δεύτερη φορά στη μνήμη που ελευθερώθηκε με την ανακύκλωση της **a*2. Φυσικά, αυτό στηρίζεται στο ότι:

- Ο "`delete`", παρά το όνομά του, δεν κάνει οποιαδήποτε διαγραφή.
- Ο "`new`", παρά το όνομά του μπορεί να μας φέρει «παλιά» πράγματα.

Παρόμοια πράγματα μπορεί να συμβούν και με τη μνήμη στοίβας:

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

void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
}

void f2()
{
   char q[5];
   cout << "from f2:  q = " << q << endl;
}

int main()
{
   f1();
   f2();
}
```

Αυτό το πρόγραμμα θα δώσει:

```
from f1:  m = abcd
from f2:  q = abcd
```

Τι έγινε εδώ; Με την κλήση της *f1()* η τοπική μεταβλητή *m* υλοποιείται σε μνήμη που δίνεται από τη στοίβα και παίρνει τιμή `"abcd"`. Με το τέλος της εκτέλεσης της *f1()* η μνήμη αυτή απελευθερώνεται. Εδώ δεν έχουμε ούτε "`new`" ούτε "`delete`"· όλα γίνονται αυτομάτως. Όταν στη συνέχεια ενεργοποιείται η *f2()*, με τον τρόπο που λειτουργεί η στοίβα, η μνήμη που είχε δοθεί στην *m* της *f1()* δίνεται στην *q* της *f2()*.

Τι κοινό έχουν τα παραπάνω παραδείγματα; *Και στις δύο περιπτώσεις πήραμε μνήμη για να κάνουμε τη δουλειά μας και μετά την ελευθερώσαμε. Αλλά τα δεδομένα που αποθηκεύτηκαν έχουν παραμείνει.*

Αν τα δεδομένα είναι «ευαίσθητα», για παράδειγμα κάποια *συνθηματικά πρόσβασης* (passwords), θα θέλαμε να έχουμε τη σιγουριά ότι όταν δεν τα χρησιμοποιούμε στο πρόγραμμά μας δεν υπάρχουν στη μνήμη του υπολογιστή. Ένας απλός τρόπος να το σιγουρέ-

ψουμε είναι να σβήσουμε πριν ανακυκλωθούν. Υπάρχει μια πάγια τεχνική για τη δουλειά αυτή: γεμίζουμε την περιοχή της μνήμης που μας ενδιαφέρει με κάποιον χαρακτήρα –συνήθως τον `'\0'`– με τη συνάρτηση της C *memset()*:[14]

```
   memset( a2, '\0', sizeof(double) );  delete a2;
```

και στην *f*1:

```
void f1()
{
   char m[] = "abcd";
   cout << "from f1:  m = " << m << endl;
   memset( m, '\0', strlen(m) );
}
```

Σημείωση: ▶
Η *memset()* έχει επικεφαλίδα:

`void* memset( void* s, int c, size_t n );`

και λειτουργεί ως εξής: βάζει *n* αντίγραφα του χαρακτήρα *c* στην περιοχή της μνήμης που αρχίζει από τη διεύθυνση *s*. Επιστρέφει την τιμή του *s*. ◀

Πάντως τα πράγματα μπορεί και να μη γίνουν όπως τα περιμένεις: Μερικοί μεταγλωττιστές, υπερβολικώς «ευφυείς», αν βρουν εντολές που αλλάζουν το περιεχόμενο μιας περιοχής της μνήμης που στη συνέχεια ανακυκλώνεται –χωρίς να μεσολαβεί χρήση του περιεχόμενου– *δεν μεταγλωττίζουν τις εντολές αλλαγής για να κάνουν το πρόγραμμα ταχύτερο!* Το ΛΣ μπορεί να έχει εργαλεία για τη λύση του προβλήματος.[15]

Η δεύτερη σύσταση έχει να κάνει με τη μνήμη συνολικώς και όχι μόνον τη δυναμική:

- Στην §16.3 μιλήσαμε για τη διαδικασία ανταλλαγών μνήμης και είπαμε ότι όλη η μνήμη που απαιτείται για το κάθε πρόγραμμα βρίσκεται κατ' αρχήν στον δίσκο. Αν λοιπόν κάποιος «κακός» ξέρει καλά το ΛΣ αλλα δεν μπορεί να «σπάσει» τις προστασίες του Συστήματος Διαχείρισης Βάσεων Δεδομένων (DBMS) μπορεί να «κλέβει» στιγμιότυπα μνήμης προγραμμάτων που χρησιμοποιούν τα δεδομένα που τον ενδιαφέρουν.
- Ορισμένα ΛΣ, όταν έχουν *μη κανονικό* τερματισμό εκτέλεσης προγράμματος δίνουν μια **απόρριψη** (περιεχόμενου της) **μνήμης** του (memory dump) σε κάποιο αρχείο, για να διευκολύνουν τον εντοπισμό του προβλήματος. Ο «κακός», που λέγαμε παραπάνω, θα μπορούσε να προκαλεί μη κανονικούς τερματισμούς προγραμμάτων και να κλέβει τα αρχεία με τα περιεχόμενα της μνήμης.

Αυτά τα προβλήματα μπορεί να αντιμετωπισθούν μόνον μέσω του ΛΣ.

16.15 Ανακεφαλαίωση

Στο πρόγραμμά μας μπορούμε να παίρνουμε μνήμη για να υλοποιήσουμε μεταβλητές ή πίνακες σύμφωνα με τις ανάγκες που προκύπτουν κατά τη διάρκεια της εκτέλεσης. Αν η *p* είναι μεταβλητή-βέλος τύπου *T** τότε:

- Με την εντολή "`p = new T`" παίρνουμε μνήμη για την υλοποίηση μιας *δυναμικής μεταβλητής* τύπου *T*. Η *p* δείχνει αυτήν τη μεταβλητή που τη χειριζόμαστε ως "`*p`".
- Η **p* είναι όπως οποιαδήποτε μεταβλητή τύπου *T* και ως τέτοια τη χειριζόμαστε.
- Όταν δεν χρειαζόμαστε την **p* την ανακυκλώνουμε με την "`delete p`". Αμέσως μετά τη "`delete p`" η **p* δεν υπάρχει και –επομένως– δεν μπορείς να τη χρησιμοποιήσεις.
- Με την "`p = new T[Π]`" ζητούμε μνήμη για έναν δυναμικό πίνακα με στοιχεία τύπου *T*. Το πλήθος τους καθορίζεται από την τιμή *n* της παράστασης *Π* που θα πρέπει να είναι ακέραιου τύπου και θετική.

[14] Θα πρέπει να έχεις δώσει "`#include <string>`" για να τη χρησιμοποιήσεις.

[15] Για παράδειγμα, στο Windows API υπάρχει η συνάρτηση *SecureZeroMemory()*.

- Τα στοιχεία του πίνακα είναι `p[0]`, `p[1]`, … `p[`*n*`-1]`. Χειριζόμαστε τον πίνακα όπως έναν συνήθη πίνακα με *n* στοιχεία τύπου *T*.
- Όταν δεν χρειαζόμαστε τον δυναμικό πίνακα τον ανακυκλώνουμε με την "**`delete p[]`**".
- Όταν ένα βέλος δεν δείχνει συγκεκριμένο στόχο –π.χ. μετά από δήλωση χωρίς αρχική τιμή ή μετά από ανακύκλωση του στόχου– βάζουμε στο βέλος τιμή "**`0`**" (ή "**`NULL`**" ή (*std*`::`)"*nullptr*").
- Η απόπειρα ανακύκλωσης ήδη ανακυκλωμένης μνήμης ("**`delete p; delete p`**") είναι σοβαρό λάθος. Η "**`delete 0`**" είναι δεκτή και δεν δημιουργεί πρόβλημα. Το ίδιο και η "**`delete[] 0`**".

Έχεις δυνατότητα να χειριστείς τη δυναμική μνήμη με τα εργαλεία της C. Αλλά:

- Μνήμη που παίρνεις με *malloc()*, *calloc()* θα πρέπει να ανακυκλώνεται με *free()* και όχι με **`delete`**.
- Μνήμη που παίρνεις με **`new`**, **`new … […]`** θα πρέπει να ανακυκλώνεται με **`delete`**, **`delete[]`** αντιστοίχως.

Ακόμη καλύτερα, αν αυτό είναι δυνατόν:

- Μην αναμειγνύεις τους δύο τρόπους στο ίδιο πρόγραμμα.

Εκτός από το «παράπτωμα» της επαναλαμβανόμενης ανακύκλωσης τα σοβαρότερα προβλήματα με τη χρήση δυναμικής μνήμης είναι τα εξής:

- Η *διαρροή μνήμης*, δηλαδή η απώλεια ελέγχου σε τμήμα δυναμικής μνήμης που παραμένει δεσμευμένη από το πρόγραμμά μας αλλά δεν μπορούμε να τη χειριστούμε διότι δεν υπάρχει βέλος που να τη δείχνει.
- Το *μετέωρο βέλος*, δηλαδή βέλος που δείχνει τμήμα δυναμικής μνήμης που έχει ήδη ανακυκλωθεί.

Ασκήσεις

Α Ομάδα

16-1 Στην §14.7.3 γράψαμε τη *swap()* για ορμαθούς της C χρησιμοποιώντας ως ενδιάμεσο ενταμιευτή μια μεταβλητή τύπου *string*. Τότε δεν ξέραμε από δυναμική μνήμη και πήραμε βοήθεια από τον τύπο *string* (που ξέρει). Τώρα, που έμαθες τη χρήση της δυναμικής μνήμης, να την ξαναγράψεις. Προσεκτικά...

16-2 Ας ξαναδούμε το πρόγραμμα με το οποίο δοκιμάζαμε τον αλγόριθμο του Horner για την τιμή πολυωνύμου (§9.4, Παράδ. 1). Στην πρώτη μορφή, είχαμε δηλώσει:

```
const int N = 20;
double a[N];
```

έναν αρκετά μεγάλο πίνακα a που θα μπορούσε να χωρέσει συντελεστές πολυωνύμου μέχρι και 19ου βαθμού. Έτσι όμως, από τη μια δεν μπορούμε να υπολογίσουμε τιμές πολυωνύμου 20ου βαθμού και από την άλλη αν έχουμε πολυώνυμα μικρού βαθμού πολλά στοιχεία του πίνακα είναι άχρηστα.

Τώρα, χρησιμοποιώντας δυναμικό πίνακα συντελεστών, μπορείς να έχεις ακριβώς αυτά που χρειάζεσαι.

16-3 Τροποποιώντας την *SList_deleteAll()*, γράψε μια *SList_clear()* που «καθαρίζει» τη λίστα –ανακυκλώνει όλους τους κόμβους αλλά όχι τον φρουρό– ώστε να μπορεί να χρησιμοποιηθεί ξανά.