κεφάλαιο

20

Κλάσεις και Αντικείμενα – Βασικές Έννοιες

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα δούμε βασικές έννοιες σχετικά με αντικείμενα και κλάσεις και με τις υλοποιήσεις τους στη C++.

Προσδοκώμενα αποτελέσματα:
Όταν θα έχεις μελετήσει αυτό το κεφάλαιο θα μπορείς να σχεδιάσεις και να υλοποιήσεις ακόμη πιο απαιτητικές κλάσεις.

Έννοιες κλειδιά:

- *μήνυμα προς αντικείμενο*
- *διεπαφή αντικειμένου*
- *απόκρυψη πληροφορίας*
- *επιφόρτωση τελεστή εκχώρησης*
- *βέλος* `this`
- *συσχετίσεις κλάσεων*

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:
Στο προηγούμενο κεφάλαιο ξεκινήσαμε με μια «γεφυρα» ανάμεσα σε αυτά που ξέραμε και στις νέες έννοιες και τεχνικές. Αυτό όμως έγινε μέσα στα πλαίσια της C++. Ο **αντικειμενοστρεφής** (object oriented) προγραμματισμός είναι κάτι ευρύτερο και πριν προχωρήσουμε στο τι κάνουμε και πώς το κάνουμε στη C++ καλό θα είναι να κάνουμε μια αφαίρεση και να δούμε μερικές έννοιες του αντικειμενοστρεφούς προγραμματισμού γενικότερα.

Ένα **αντικείμενο** (object) είναι ένα σύνολο **λειτουργιών** (operations) που δρούν πάνω σε μια κοινή **κατάσταση** (state). Μπορείς να καταλάβεις την κατάσταση ως το σύνολο τιμών μεταβλητών που βρίσκονται μέσα στο αντικείμενο και καλούνται **μεταβλητές στιγμιοτύπου** (instance variables) Η κατάσταση είναι κρυμμένη από τον έξω κόσμο. Ο έξω κόσμος έχει πρόσβαση στην κατάσταση μόνο μέσω των λειτουργιών. Μπορείς να καταλάβεις τις λειτουργίες ως συναρτήσεις που καλούνται **μέθοδοι** (methods). Το σύνολο των λειτουργιών καθορίζουν τη **διεπαφή** (interface) του και τη **συμπεριφορά** (behavior) του. (Wegner 1990)

Και τι είναι η **κλάση** (class); Η κλάση είναι ένα περίγραμμα, ένα σχέδιο με βάση το οποίο δημιουργούνται αντικείμενα. Έχει τις προδιαγραφές της κοινής συμπεριφοράς όλων των αντικειμένων αυτής της κλάσης. Οι μεταβλητές-μέλη της κλάσης είναι **δυνάμει** (potential) μεταβλητές σε αντιδιαστολή με αυτές του αντικειμένου που είναι **πραγματικές** (actual). Στην κλάση περιγράφονται και οι λειτουργίες με τις οποίες θα εξοπλισθεί και κάθε αντικείμενο-στιγμιότυπο της κλάσης. Περιγράφεται επίσης και η λειτουργία **δημιουργίας στιγμιοτύπου** (instance creation).

Για παράδειγμα, ο ορισμός

```
class Date
{ // I: (dYear > 0) && (0 < dMonth <= 12) &&
  //    (0 < dDay <= lastDay(dYear, dMonth))
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth()
                       const {  return static_cast<unsigned int>( dMonth );  }
   unsigned int getDay() const {  return static_cast<unsigned int>( dDay );  }
   void load( istream& bin );
   void save( ostream& bout ) const;
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
}; // Date
```

είναι ένα περίγραμμα που δείχνει πώς θα δημιουργούνται τα αντικείμενα τύπου *Date*. Η αναλλοίωτη *I* είναι μια συνθήκη με την οποίαν συμμορφώνεται η κατάσταση του κάθε αντικειμένου κλάσης *Date*. Οι μεταβλητές `Date::dYear`, `Date::dMonth`, `Date::dDay` είναι οι δυνάμει μεταβλητές της κλάσης. Η λειτουργία δημιουργίας στιγμιοτύπου περιγράφεται από τον δημιουργό (στην πραγματικότητα τέσσερις δημιουργοί).

Οι `Date::lastDay()` και `Date::leapYear()` τι είναι; Αυτό που ήδη είπαμε: βοηθητικές συναρτήσεις που μας χρειάζονται για την υλοποίηση των μεθόδων. Τις κρατούμε «μυστικές» όπως άλλωστε και όλα τα συστατικά της υλοποίησης.

Μετά τη δήλωση:

```
   Date d1( 2007, 5, 31 ), d2( 2004, 2, 29 );
```

έχουμε δύο αντικείμενα, τα *d1* και *d2*.

Η κατάσταση του πρώτου περιγράφεται από τις τιμές των (πραγματικών) μεταβλητών περίπτωσης *d1.dYear*, *d1.dMonth* και *d1.dDay* που είναι 2007, 5 και 31 αντιστοίχως. Κρατούμε αυτές τις μεταβλητές κρυμμένες αφού δηλώσαμε τις αντίστοιχες δυνάμει μεταβλητές σε περιοχή `private`.

Η κατάσταση του δεύτερου περιγράφεται από τις τιμές των (επίσης κρυμμένων) μεταβλητών στιγμιοτύπου *d2.dYear*, *d2.dMonth* και *d2.dDay* που είναι 2004, 2 και 29 αντιστοίχως.

Στην κατάσταση του *d1* δρουν οι μέθοδοι: *d1.getYear*`()`, *d1.getMonth*`()`, *d1.getDay*`()`, *d1.load*`(`*istream*`&)` και *d1.save*`(`*ostream*`&)`. Αυτές οι μέθοδοι βγάζουν προς τον έξω κόσμο πληροφορίες σχετικώς με την κατάσταση του *d1*. Πέρα από αυτές όμως υπάρχει και μια «κρυφή» αλλά πολύ ουσιαστική μέθοδος που επιτρέπει στον έξω κόσμο να αλλάξει την κατάσταση του αντικειμένου: ο *τελεστής εκχώρησης* (“`=`”). Για τα αντικείμενα τύπου *Date*, αυτή η μέθοδος έχει ορισθεί αυτομάτως από τον μεταγλωττιστή.

Όλο το κομμάτι που είναι `public` καθορίζει τη συμπεριφορά του κάθε αντικειμένου. Αυτό είναι το τμήμα *διεπαφής* του κάθε αντικειμένου κλάσης *Date*.

Στο τμήμα διεπαφής καθορίζονται τα είδη των **μηνυμάτων** (messages) που αναγνωρίζει, που μπορεί να δέχεται, ένα αντικείμενο από τον έξω κόσμο: είτε από το πρόγραμμα-πελάτη είτε από άλλα αντικείμενα που «ζούν» στο πρόγραμμα. Το αντικείμενο ανταποκρίνεται στα μηνύματα αυτά είτε αλλάζοντας την κατάστασή του είτε δίνοντας πληροφορίες σχετικώς με αυτήν. Για παράδειγμα, στέλνοντας στο *d1* το μήνυμα:

```
d1.load( bin );
```

του ζητούμε να αλλάξει την κατάστασή του διαβάζοντας νέες τιμές για τα μέλη από το ρεύμα *bin*.

Αν έχουμε δηλώσει:

```
Battery btr;
```

μπορούμε να στείλουμε τα μηνύματα *btr*`.`*powerDevice*`(double,double,bool&)`, *btr*`.`*maxTime*`( double )`, *btr*`.`*reCharge*`()`. Στέλνοντας στο *btr* το μήνυμα:

```
btr.powerDevice( 15*60, 4, ok );
```

ζητούμε από το *btr* να αλλάξει την κατάστασή του, αν αυτό είναι δυνατόν. Μέσω της *ok* το αντικείμενο μας γνωστοποιεί αν την άλλαξε.

Με το μήνυμα `btr.maxTime(8)` ζητούμε από το *btr* μια πληροφορία που έχει σχέση με την κατάστασή του.

Ένα αντικείμενο δεν ξέρει ποιος του στέλνει ένα μήνυμα που λαμβάνει, εκτός αν ο αποστολέας περιλαμβάνει τέτοια πληροφορία μέσα στο μήνυμα.

Από εδώ και πέρα . . .

Από εδώ και πέρα θα δούμε πώς γίνονται τα παραπάνω στη C++. Είδαμε ήδη μερικά στο προηγούμενο κεφάλαιο.

Θα δεις ότι η δουλειά μας έχει να κάνει κατά κύριο λόγο με τη διεπαφή:

- Τι θα βάλουμε εκεί και
- Πώς θα το υλοποιήσουμε.

Με το κλειστό μέρος δεν θα ασχοληθούμε; Όχι και πολύ αφού δεν έχει και πολλά πράγματα να βάλουμε εκεί:

- Όπως είδαμε, εκεί κρύβουμε τις βοηθητικές συναρτήσεις.
- Αργότερα θα δούμε ότι εκεί «κρύβουμε» μερικές μεθόδους επειδή θέλουμε να τις «αχρηστεύσουμε».

20.1 Άλλο Ένα Παράδειγμα: *BString*

Πριν προχωρήσουμε παρακάτω θα κάνουμε μια εισαγωγή σε άλλο ένα παράδειγμα κλάσης που θα χρησιμοποιούμε στη συνέχεια. Πρόκειται για την κλάση *BString* που θα μιμείται την `std::string`. Φυσικά, δεν θα υλοποιήσουμε την *BString* όπως ακριβώς είναι υλοποιημένη η *string*. Μάλλον θα κάνουμε την *BString* έτσι ώστε τα αντικείμενά της να συμπεριφέρονται όπως (περίπου) αυτά της *string*.

Για να μπορούμε να κρατούμε ως τιμή οποιονδήποτε ορμαθό θα χρησιμοποιήσουμε *δυναμικό πίνακα* με στοιχεία τύπου `char`.

```
char*  bsData;
```

Αυτό δεν σημαίνει κατ' ανάγκην ότι θα παίρνουμε πάντοτε τόση μνήμη όση μας χρειάζεται για να αποθηκεύσουμε την τιμή που θέλουμε.[1] Γιατί όχι; Ας πούμε ότι το πρόγραμμά μας δημιουργεί, με κάποιον τρόπο, 100 τιμές τύπου `char` (χαρακτήρες) και τους αποθηκεύει έναν προς ένα σε μια μεταβλητή τύπου *BString*. Ας πούμε ότι για κάθε χαρακτήρα που έρχεται ακολουθούμε τον αλγόριθμο της *renew()*: παίρνουμε νέα μνήμη, αντιγράφουμε το περιεχόμενο της παλιάς στη νέα, ανακυκλώνουμε την παλιά.

- Όταν έλθει ο *δεύτερος* χαρακτήρας παίρνουμε πίνακα δύο χαρακτήρων, *αντιγράφουμε έναν χαρακτήρα*, ανακυκλώνουμε πίνακα ενός χαρακτήρα.
- Όταν έλθει ο *τρίτος* χαρακτήρας παίρνουμε πίνακα τριών χαρακτήρων, *αντιγράφουμε δύο χαρακτήρες*, ανακυκλώνουμε πίνακα δύο χαρακτήρων.
- Όταν έλθει ο *τέταρτος* χαρακτήρας παίρνουμε πίνακα τεσσάρων χαρακτήρων, *αντιγράφουμε τρεις χαρακτήρες*, ανακυκλώνουμε πίνακα τριών χαρακτήρων, κ.ο.κ.

Συνολικώς: θα πάρουμε μνήμη 100 φορές, θα ανακυκλώσουμε μνήμη 99 φορές και θα κάνουμε 1+2+ ... +99 = 4950 αντιγραφές χαρακτήρων.

Αν, κάθε φορά που χρειάζεται να πάρουμε μνήμη, αντί να την αυξάνουμε κατά 1 την αυξάνουμε κατά 50, θα πάρουμε μνήμη –συνολικώς– 2 φορές, θα ανακυκλώσουμε μνήμη 1 φορά και θα κάνουμε 50 αντιγραφές χαρακτήρων.

Για να αποφύγουμε λοιπόν το διαρκές πάρε-δώσε με τον σωρό και τις αντιγραφές θα παίρνουμε κάτι περισσότερο.[2] Επομένως θα χρειαστούμε άλλο ένα μέλος

```
size_t bsReserved;
```

που θα μας δίνει το πλήθος θέσεων του πίνακα που έχουν παραχωρηθεί στο αντικείμενο.

Θα κάνουμε λοιπόν την εξής συμφωνία: *Οποτε έχουμε να πάρουμε μνήμη θα παίρνουμε πολλαπλάσιο των* 16 *ψηφιολέξεων*. Γιατί 16; Δεν υπάρχει κάποιος ιδιαίτερος λόγος. Εδώ θα κάνουμε απλώς επίδειξη ορισμένων τεχνικών και τίποτε παραπάνω. Αλλά, σε άλλες περιπτώσεις, όταν έχεις αντικείμενα με δυναμικούς πίνακες, μπορείς να βρεις πόσο είναι ένα καλό «κβάντο αύξησης».

Όταν αποθηκεύουμε μια τιμή (ορμαθό), σε ένα αντικείμενο κλάσης *BString*, πώς θα ξέρουμε πόσες θέσεις του δυναμικού πίνακα καταλαμβάνει;

- Ένας τρόπος είναι να βάζουμε, ως φρουρό, ένα `'\0'` στο τέλος του ορμαθού και να μετρούμε μέχρι να το βρούμε. Αυτό μας είναι χρήσιμο και σε κάτι άλλο: Η *string* έχει τη μέθοδο *c_str()* που επιστρέφει έναν ορμαθό χαρακτήρων της C, δηλαδή ένα βέλος προς την αρχή ενός πίνακα τύπου `char` με `'\0'` στο τέλος. Αν περιλάβουμε το `'\0'` στο τέλος η υλοποίησή της είναι πολύ απλή.

---

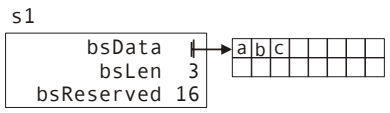
[1] Ξαναδές και αυτά που είπαμε στο Παράδ. 2 της §16.13.

[2] Η υλοποίηση της C++ χρησιμοποιεί έναν πιο πολύπλοκο τρόπο για τη διαχείριση της δυναμικής μνήμης. Ένα χαρακτηριστικό του, που πρέπει να αναφέρουμε, είναι το εξής: αντί να αυξάνει την «καπαρωμένη» μνήμη κατά ένα σταθερό «κβάντο» κάθε φορά την *διπλασιάζει*. Έτσι κάνει λιγότερες αντιγραφές (δες την §16.13.3).

- Ένας άλλος τρόπος είναι ο εξής: κρατούμε σε ένα χωριστό μέλος, *bsLen*, το μήκος του κειμένου που αποθηκεύουμε στο αντικείμενο.
- Ένας τρίτος τρόπος είναι να χρησιμοποιήσουμε δύο βέλη:

```
char* bsData;
char* bsEnd;
```

που θα δείχνουν την αρχή και το τέλος του κειμένου.



Σχ. 20-1 Η παράσταση ενός αντικειμένου *s1* κλάσης *BString* (*bsIncr* = 16.)

Πρόσεξε ότι ο δεύτερος και ο τρίτος τρόπος μας επιτρέπουν να έχουμε και χαρακτήρες `'\0'` στην τιμή που αποθηκεύουμε.

Θα χρησιμοποιήσουμε τον δεύτερο τρόπο που είναι ο πιο απλός και βολικός (Σχ. 20-1).

```
class BString
{
public:
// ...
private:
   enum { bsIncr = 16 };
   char*  bsData;
   size_t bsLen;
   size_t bsReserved;
}; // BString
```

Τι είναι εκείνο το *bsIncr*; Όπως βλέπεις είναι μια ακέραιη σταθερά με τιμή 16. Εεπειδή, όπως είπαμε θα παίρνουμε δυναμική μνήμη σε «φέτες» των 16 ψηφιολέξεων και αυτό θα φαίνεται στις υλοποιήσεις των μεθόδων, προτιμούμε να δώσουμε ένα όνομα στη σταθερά μας για να μην εμφανίζεται ως «μαγική σταθερά».

Με βάση αυτά η αναλλοίωτη –μέχρι τώρα– διαμορφώνεται ως εξής:

(0 <= *bsLen* < *bsReserved*) && (*bsReserved* % *bsIncr* == 0)

Αφού *bsLen* < *bsReserved* υπάρχει πάντοτε θέση για τον φρουρό (`'\0'`) όταν μας χρειαστεί.

Η κλάση εξαιρέσεων θα είναι κάπως έτσι:

```
struct BStringXptn
{
   enum { . . . };
   char funcName[100];
   int  errorCode;
   int  errorValue;
   BStringXptn( char* mn, int ec, int ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BStringXptn
```

20.1.1 Οι Απλές Μέθοδοι

Δες πόσο εύκολα υλοποιείται η *c_str*:

```
   const char* c_str() const
   {  bsData[bsLen] = '\0';  return bsData;  }
```

Βάζουμε τον φρουρό και επιστρέφουμε το βέλος. Το "`const`" *στην αρχή* είναι απαραίτητο διότι αλλιώς, όποιος πάρει το βέλος *bsData*, μπορεί να αλλάζει τον ορμαθό που είναι αποθηκευμένος.

Προσοχή όμως! Αν μέσα στο κείμενο υπάρχει ο χαρακτήρας `'\0'` και προσπαθήσουμε να επεξεργαστούμε αυτό που μας δίνει η *c_str* με τις συναρτήσεις *str…* η επεξεργασία θα τελειώνει στον πρώτο `'\0'`.

Παρατήρηση: ▶

Το "`const`" *στο τέλος* δεν μας απαγορεύει να βάλουμε τιμή `'\0'` στο τέλος του κειμένου (`bsData[bsLen]`). Μας απαγορεύει να αλλάξουμε τις τιμές των μελών *bsData* (βέλος), *bsLen*, *bsReserved*. Για να βάλουμε απαγόρευση αλλαγής του στόχου του *bsData* θα έπρεπε να είχαμε δηλώσει

```
const char* bsData;
```

Αλλά αν κάνουμε κάτι τέτοιο, κάθε φορά που θέλουμε να αλλάξουμε το κείμενο θα πρέπει να κάνουμε τυποθεώρηση **const**. Υπερβολές... ◀

Εύκολα μπορούμε να υλοποιήσουμε και τη μέθοδο *length()*:

```
size_t length() const {  return bsLen;  }
```

όπως και την *empty()*:

```
bool empty() const {  return ( bsLen == 0 );  }
```

Να σημειώσουμε ότι:""

- "*c_str*" είναι αυτό που θα λέγαμε "*getData*" και
- "*length*" αυτό που θα λέγαμε "*getLen*".
- Η *empty()* είναι ένα **κατηγόρημα** (predicate). Θα το ονομάζαμε "*isEmpty*" αλλά κρατούμε την ονοματολογία του τύπου *string*.

Παρατήρηση: ▶

Η *string* έχει και μια άλλη μέθοδο, συνώνυμη της *length()*, τη *size()*. Να την υλοποιήσουμε και αυτήν; Απλό:

```
size_t size() const {  return bsLen;  }
```

Σωστό! Αλλά όχι καλό. Δες μια άλλη επιλογή:

```
size_t size() const {  return length();  }
```

Μια άλλη επιλογή είναι: να ορίσουμε πρώτα τη *size()* και μετά τη *length()* με βάση τη *size()*. Πάντως, όποτε έχεις συνώνυμες μεθόδους

- Πρώτα ορίζεις τη μια αυτές.
- Μετά ορίζεις τις συνώνυμες με βάση αυτήν που ορίστηκε.

Έτσι, αν χρειαστεί να κάνεις κάποια αλλαγή αυτή θα πρέπει να γίνει σε ένα μόνο σημείο. Θα επανέλθουμε…◀

Ας πάμε τώρα να δούμε έναν δημιουργό. Όταν δηλώσουμε:

```
BString s0;
```

το *s0* θα έχει ως τιμή τον ορμαθό μηδενικού μήκους.

```
BString::BString()
{
   try {  bsData = new char[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BString::BString
```

Κατά τη συνήθειά μας, για να έχουμε ενιαία διαχείριση των εξαιρέσεων από τα αντικείμενα της κλάσης, πιάνουμε τη *bad_alloc* και ρίχνουμε μια *BStringXptn* (*allocFailed*).

Μια άλλη περίπτωση είναι η εξής: δίνουμε ως αρχική τιμή έναν ορμαθό της C (πίνακα `char` με φρουρό `'\0'`):

```
BString s1( "abc" );
```

Η περίπτωση αυτή καλύπτεται από τον δημιουργό:

```
BString( const char* rhs );
```

Η υλοποίησή του γίνεται ως εξής:

```
BString::BString( const char* rhs )
{
```

```
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   try {  bsData = new char [bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BString
```

Αν γράψουμε την επικεφαλίδα του δεύτερου ως εξής:

```
   BString( const char* rhs="" );
```

ο δεύτερος γίνεται «2 σε 1» και ο πρώτος δεν μας χρειάζεται.

Εδώ πρόσεξε τα εξής:

1. Για να υπολογίσουμε το μήκος του ορίσματος χρησιμοποιούμε μια *εσωτερική μέθοδο*:[3]

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

Πρόσεξε ότι στο μήκος δεν περιλαμβάνεται η θέση για το `'\0'`. Γιατί δεν χρησιμοποιούμε τη *strlen*; Γενικώς δεν χρησιμοποιούμε τις συναρτήσεις διαχείρισης ορμαθών της C. Αργότερα θα καταλάβεις τον λόγο.

2. Πόσες ψηφιολέξεις μνήμης παίρνουμε; Όπως είπαμε θα παίρνουμε πάντοτε μνήμη που είναι ακέραιο πολλαπλάσιο των *bsIncr* == 16 ψηφιολέξεων. Εδώ θα πρέπει να πάρουμε μνήμη ίση με το πρώτο πολλαπλάσιο των 16 ψηφιολέξεων που είναι μεγαλύτερο από το *bsLen*+1 (για τον φρουρό).

3. Ο *BStringXptn*::*allocFailed* είναι ο πρώτος κωδικός σφάλματος που χρησιμοποιούμε. Τώρα δηλαδή θα έχουμε:

```
   enum { allocFailed };
```

20.1.2 … Και η Μέθοδος *at()*

Ας δούμε και μία ακόμη απλή μέθοδο, την *at*, για την οποίαν έχουμε αφήσει και κάποια «χρέη» από παλιά. Γράφαμε στην §10.10: «*Αυτό που γράψαμε παραπάνω μπορεί να γραφεί και ως εξής:*

```
   if ( s3.at(0) == 'κ' )  s3.at( 0 ) = 'K';
```

Αυτός ο τρόπος είναι ασφαλέστερος από αυτόν με τον δείκτη, διότι αν κάνεις λάθος και βγεις έξω από τα όρια της μεταβλητής σου θα ειδοποιηθείς, όπως θα μάθουμε αργότερα.» Τώρα πια μπορούμε να πούμε ότι θα ειδοποιηθούμε με μια εξαίρεση κλάσης *out_of_range* που δηλώνεται στο `stdexcept`. Δοκίμασε το:

```
#include <iostream>
#include <stdexcept>
#include <string>

using namespace std;

int main()
{
   string s3( "abc" );

   try
   {
      s3.at( 7 ) = '@';
```

[3] Κάτι σου θυμίζει; Είναι η δεύτερη μορφή της *myStrLen* από το παράδ. 1 της §12.3.3.

```
   }
   catch( out_of_range& x )
   {
      cout << "out_of_range" << endl;
   }
}
```

Τώρα πρόσεξε το εξής: αν η *s3* δηλωθεί ως αντικείμενο κλάσης *BString*, η "`s3.at(m)`" θα πρέπει να μας δίνει όχι απλώς την τιμή του `s3.bsData[m]` αλλά το ίδιο το στοιχείο ώστε να μπορούμε να αλλάξουμε την τιμή του. Πώς γίνεται αυτό; Από την §13.4 αντιγράφουμε: «*Βάζοντας ως τύπο επιστροφής της συνάρτησης "*`int&`*" η συνάρτηση επιστρέφει το στοιχείο με τη μέγιστη τιμή (τιμή-l) και όχι απλώς την τιμή του.*» Αυτή είναι η λύση για το πρόβλημά μας:

```
char& BString::at( int k ) const
{
   if ( k < 0 || bsLen <= k )
      throw BStringXptn( "at", BStringXptn::outOfRange, k );
   return bsData[k];
} // BString::at
```

Η συνάρτηση επιστρέφει τιμή τύπου "`char&`" –στην περίπτωσή μας το στοιχείο `bsData[k]`– και όχι "`char`".

Η *at* μπορεί να ρίξει εξαίρεση κλάσης *BStringXptn* (με κωδικό *outOfRange*).

Να επιστήσουμε πάντως την προσοχή σου στο εξής: Με την *at* δίνουμε στο πρόγραμμα που φιλοξενεί το αντικείμενο δυνατότητα να αλλάζει την τιμή του. Στη συνέχεια θα δεις ότι πολύ συχνά θα γράφουμε μεθόδους που βγάζουν τιμή με τύπο αναφοράς απλώς και μόνο για να αποφεύγουμε μια αντιγραφή. Σε τέτοιες περιπτώσεις μάλλον θα χρειάζεται και κάποιο "`const`".

20.1.3 Ο Καταστροφέας

Τώρα έχουμε και ένα άλλο πρόβλημα: Ας πούμε ότι σε μια συνάρτηση έχουμε δηλώσει και χρησιμοποιήσει κάποιο αντικείμενο κλάσης *BString*. Τι θα γίνει όταν τελειώσει η εκτέλεση της συνάρτησης; Θα «καταστραφούν» αυτομάτως τα μέλη αλλά δεν θα ανακυκλωθεί η (δυναμική) μνήμη –που δείχνει το μέλος *bsData*– όπου έχουμε αποθηκεύσει το κείμενο. Αυτό θα το φροντίσει ένας **καταστροφέας** (destructor) που θα γράψουμε εμείς. Ενώ, όπως είδες, μπορεί να έχουμε πολλούς δημιουργούς έχουμε έναν και μόνο καταστροφέα που το όνομά του είναι "`~`", *όνομα κλάσης*:

```
BString::~BString()
{
   delete[] bsData;
} // BString::~BString
```

Για καταστροφείς, όπως και για δημιουργούς, θα μιλήσουμε εκτενώς στη συνέχεια.

20.2 Ο Δημιουργός Αντιγραφής

Ας πούμε ότι δίνουμε:

```
      BString s1( "abc" );
      cout << "s1: " << s1.c_str() << endl;

      BString s2( s1 ), s3( s1.c_str() );
      cout << "s2: " << s2.c_str() << endl;
      cout << "s3: " << s3.c_str() << endl;
```

Αποτέλεσμα:

```
s1: abc
s2: abc
```

```
s3: abc
```

Παρατήρηση: ▶

Αν προτιμάς να γράφεις

```
BString s2 = s1,
        s3 = s1.c_str();
```

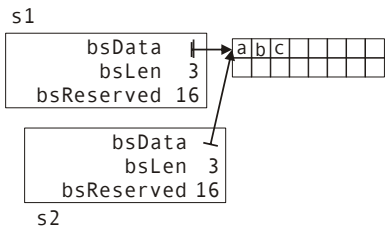
κανένα πρόβλημα· θα πάρεις τα ίδια ακριβώς απότελέσματα.◀

Μέχρι εδώ κανένα πρόβλημα. Να σημειώσουμε μόνον ότι η *s3* παίρνει τιμή με τον ίδιο τρόπο που παίρνει και η *s1*. Πρόσεξε τώρα τη συνέχεια:

```
   s1.at( 1 ) = 'v';
   cout << "s1: " << s1.c_str() << endl;
   cout << "s2: " << s2.c_str() << endl;
   cout << "s3: " << s3.c_str() << endl;
```

Αποτέλεσμα:

```
s1: avc
s2: avc
s3: abc
```



Σχ. 20-2 Η παράσταση της κατάστασης των *s1* και *s2* μετά τη "`BString s2(s1));`"

Εμείς αλλάξαμε την τιμή του *s1*· γιατί άλλαξε και η τιμή του *s2*; Δες το Σχ. 20-2 για να καταλάβεις: Ο μεταγλωττιστής μη έχοντας άλλες οδηγίες –ειδικές για την περίπτωση αυτή– αντίγραψε στα μέλη του νέου αντικειμένου (*s2*) τις τιμές των αντίστοιχων μελών του *s1*. Το πρόβλημα δημιουργείται από την αντιγραφή του βέλους `s1.bsData` στο `s2.bsData` που κάνει τα δύο αντικείμενα να έχουν ως τιμή το ίδιο κείμενο.

Να θυμίσουμε ότι αυτό το πρόβλημα –αλλά για τον τελεστή εκχώρησης– το πρωτοείδαμε στην §16.8 χωρίς να το λύσουμε. Τώρα θα το λύσουμε. Πώς; Με τον ίδιο τρόπο που κάνουμε αντιγραφή χωρίς πρόβλημα στην *s3*: Θα ορίσουμε ειδικώς για τον ορισμό αντικειμένου με αρχική τιμή άλλο αντικείμενο του ίδιου τύπου έναν δημιουργό, που λέγεται **δημιουργός αντιγραφής** (copy constructor), που θα παίρνει για το νέο αντικείμενο-αντίγραφο (στο παράδειγμά μας: *s2*) όση δυναμική μνήμη έχει το πρωτότυπο (στο παράδειγμά μας: *s1*):

```
BString::BString( const BString& rhs )
{
   bsReserved = rhs.bsReserved;

   try {  bsData = new char[bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BString
```

Για τον δημιουργό αντιγραφής θα τα πούμε στο επόμενο κεφαλαιο. Εδώ θα σημειώσουμε τα εξής:

- Τα μέλη του αντικειμένου που δημιουργείται (στο παράδειγμά μας: *s2*) είναι *bsData*, *bsLen* και *bsReserved*, ενώ *rhs* είναι το πρωτότυπο (στο παράδειγμά μας: *s1*) με μέλη *rhs*.*bsData*, *rhs*.*bsLen* και *rhs*.*bsReserved*.
- Γιατί δεν κάνουμε έναν έλεγχο για το πόση μνήμη χρειάζεται; Μπορεί να χρειάζεται λιγότερη από *rhs*.*bsReserved*. Κάνε το έτσι αν θέλεις. Το δικό μας σκεπτικό είναι απλό: «το αντίγραφο πρέπει να είναι (πιστό) αντίγραφο.»

20.3 Η Πρόσβαση στα Μέλη "private"

Πριν προχωρήσουμε στο παρεμφερές πρόβλημα του τελεστή εκχώρησης θα πρέπει να επισημάνουμε το εξής: Το νέο αντικείμενο έχει πρόσβαση στα μέλη του πρωτότυπου (*rhs*) παρ' όλο που αυτά είναι σε περιοχή **private**. Γενικώς:

- ***Όλες οι μέθοδοι ενός αντικειμένου κλάσης T έχουν προσβαση σε όλα τα μέλη οποιουδήποτε άλλου αντικειμένου κλάσης T.***

20.4 Ο Τελεστής Εκχώρησης "="

Αφού δηλώσουμε:

```
BString s1( "abc" ), s2;
```

ζητούμε:

```
cout << "s1: " << s1.c_str() << endl;

s2 = s1;
cout << "s2: " << s2.c_str() << endl;

s1.at( 1 ) = 'v';
cout << "s1: " << s1.c_str() << endl;
cout << "s2: " << s2.c_str() << endl;
```

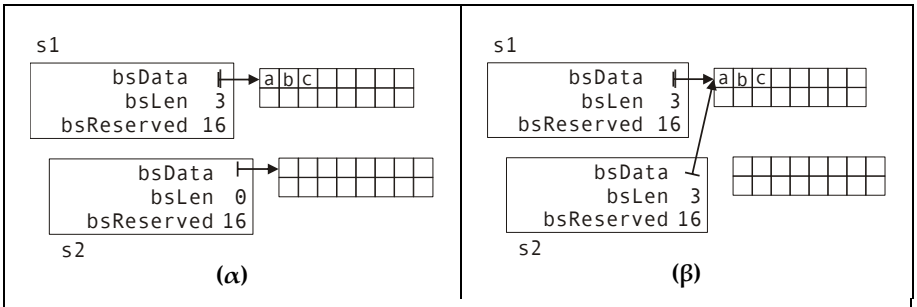
Αποτέλεσμα:

```
s1: abc
s2: abc
s1: avc
s2: avc
```

Δεν είναι σωστό· αυτό όμως –μετά από αυτά που είδαμε στην προηγούμενη παράγραφο– δεν είναι έκπληξη. Όπως καταλαβαίνεις έχουμε το ίδιο πρόβλημα αλλά σε χειρότερη μορφή: τώρα έχουμε και διαρροή μνήμης (Σχ. 20-3).

- Και εδώ έχουμε αντιγραφή στα μέλη του αντικειμένου *s2* των τιμών των αντίστοιχων μελών του *s1*.
- Και εδώ έχουμε το πρόβλημα από την αντιγραφή του βέλους `s1.bsData` στο `s2.bsData` που κάνει τα δύο αντικείμενα να έχουν ως τιμή το ίδιο κείμενο. Αλλά τώρα το `s2.bsData` έδειχνε τη δυναμική μνήμη που ήδη είχε πάρει το *s2*. Αυτή η μνήμη παραμένει δεσμευμένη από το πρόγραμμά μας αλλά δεν έχουμε εργαλείο να τη χειριστούμε.

Μια και ο τελεστής εκχώρησης είναι απαραίτητος για τον τύπο *BString* θα πρέπει να τον επιφορτώσουμε ώστε να δουλεύει σωστά. Πριν προχωρήσεις ξαναδιάβασε αυτά που λέγαμε για την επιφόρτωση των τελεστών εκχώρησης στην §14.6.3.



Σχ. 20-3 Στο (α) βλέπουμε μια παράσταση της κατάστασης των *s1* και *s2* μετά την δήλωση των αντικειμένων. Στο (β) βλέπουμε την κατάσταση των *s1* και *s2* μετά την εντολή εκχώρησης. Όπως φαίνεται, δεν υπάρχει βέλος για τον χειρισμό της δυναμικής μνήμης που έχει δεσμευθεί για το *s2*.

Αν έχουμε:

```
BString s1( "abc" ), s2, s3;
```

η

```
s2 = s1;
```

θα πρέπει να δίνει ως τιμή στο *s2* την τιμή του *s1* και να επιστρέφει αυτήν την νέα τιμή του *s2* ώστε, αν θέλουμε, να τη χρησιμοποιήσουμε· για παράδειγμα θα μπορούσαμε να δώσουμε: "`s3 = (s2 = s1)`". Πώς θα πρέπει να κάνουμε την επιφόρτωση;

Μην προσπαθήσεις να εφαρμόσεις αυτά που λέμε στην §14.6.4 (σε επόμενο κεφάλαιο θα δώσουμε νέες οδηγίες): Ο τελεστής εκχώρησης επιφορτώνεται ως *μέθοδος*:

```
BString& operator=( const BString& rhs );
```

Τι θα κάνει αυτή η μέθοδος;

```
«Καθάρισε» την παλιά τιμή
Πάρε τη μνήμη που χρειάζεσαι
Αντίγραψε την τιμή του rhs
Επίστρεψε ως τιμή το αντικείμενο
```

Και αν η «*Πάρε τη μνήμη που χρειάζεσαι*» αποτύχει; Στην περίπτωση αυτή

- θα ρίξουμε εξαίρεση
- θα πρέπει να διασφαλίσουμε ότι το αντικείμενό μας κρατάει την τιμή που έχει (όποια και αν είναι αυτή.)

Επομένως θα πρέπει να αλλάξουμε το σχέδιο: Πρώτα παίρνουμε τη μνήμη και μετά «καθαρίζουμε» την παλιά τιμή.

```
Πάρε τη μνήμη που χρειάζεσαι
if ( απέτυχες ) throw …
«Καθάρισε» την παλιά τιμή
Αντίγραψε την τιμή του rhs
return το αντικείμενο
```

Πρόσεξε τώρα κάτι ιδιαίτερο: Τι θα γίνει αν δοθεί η (φαινομενικώς ανόητη αλλά καθ' όλα νόμιμη) εντολή "`s1 = s1`"; Εδώ, *bsData* και *rhs*`.bsData` είναι το ίδιο πράγμα. Όταν έλθει η ώρα να αντιγράψεις τον πίνακα στον εαυτό του ο πίνακας έχει ήδη ανακυκλωθεί! Φυσικά, αν πρέπει να εκτελεσθεί αυτή η παράξενη «αυτοεκχώρηση», δεν χρειάζεται να κάνουμε οτιδήποτε από τα παραπάνω· αρκεί να επιστρέψουμε ως τιμή το αντικείμενο:

```
if ( !αυτοεκχώρηση )
{
   Πάρε τη μνήμη που χρειάζεσαι
   if ( απέτυχες ) throw …
   «Καθάρισε» την παλιά τιμή
   Αντίγραψε την τιμή του rhs
}
return το αντικείμενο
```

Υπάρχουν όμως δύο ανοικτά προβλήματα:

- Πώς καταλαβαίνουμε ότι έχουμε «αυτοεκχώρηση»;
- Πώς υλοποιείται η ψευδοεντολή «***return*** *το αντικείμενο*»;

Η λύση και στα δύο προβλήματα δίνεται με το βέλος **this**, για το οποίο μιλούμε στην επόμενη παράγραφο. Κάθε αντικείμενο έξοπλίζεται (αυτομάτως) με ένα βέλος, το **this** *που δείχνει το ίδιο το αντικείμενο*. Έτσι:

- Καταλαβαίνουμε ότι έχουμε «αυτοεκχώρηση» αν ισχύει η `&rhs == this`.
- Η ψευδοεντολή «***return*** *το αντικείμενο*» υλοποιείται ως:

```
return *this;
```

Για να παρακολουθήσεις τη μετάφραση του σχεδίου σε C++ πάρε υπόψη σου το εξής: Ο τελεστής επιφορτώνεται με μέθοδο του «αριστερού μέλους» της εκχώρησης. Έτσι, η "`s2 = s1`" μπορεί να γραφεί και ως "`s2.operator=(s1)`".

- Η «***if*** *(* **!***αυτοεκχώρηση)*» γίνεται:

```
if ( &rhs != this )
```

- Για να μεταφράσουμε τις «*Πάρε τη μνήμη που χρειάζεσαι*» και «***if*** *(απέτυχες)* ***throw*** *…*» γράφουμε:

```
   char* tmp;
   try {  tmp = new char[rhs.bsReserved];  }
   catch( bad_alloc& )
   { throw BStringXptn( "operator=", BStringXptn::allocFailed ); }
```

Εδώ πρόσεξε ότι παίρνουμε τη μνήμη με ένα βοηθητικό βέλος (*tmp*). Ακόμη, κατά τη συνήθειά μας, βάλαμε εντολές που πιάνουν πιθανή εξαίρεση *bad_alloc* και ρίχνουν δική μας *BStringXptn* (*allocFailed*). Οι εντολές που ακολουθούν δεν θα εκτελεσθούν αν δεν παραχωρηθεί η μνήμη που ζητούμε αφού στην περίπτωση αυτή θα ριχτεί εξαίρεση.

- Η «*"Καθάρισε" την παλιά τιμή*» γίνεται

```
   delete[] bsData;
```

- Η «*Αντίγραψε την τιμή του rhs*» γίνεται

```
   bsData = tmp;
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
```

Και η μέθοδος που επιφορτώνει τον τελεστή "=" για την κλάση μας:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )  // if ( !αυτοεκχώρηση )
   {
   // Πάρε τη μνήμη που χρειάζεσαι
      char* tmp;
      try {  tmp = new char[rhs.bsReserved];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "operator=", BStringXptn::allocFailed );  }
   // "Καθάρισε" την παλιά τιμή
      delete[] bsData;
   // Αντίγραψε την τιμή του rhs
      bsData = tmp;
      bsReserved = rhs.bsReserved;
      for ( int k(0); k < rhs.bsLen; ++k ) bsData[k] = rhs.bsData[k];
      bsLen = rhs.bsLen;
   }
   return *this;
} // BString::operator=
```

Αν, μετά την επιφόρτωση, δοκιμάσουμε το πρόγραμμά μας παίρνουμε:

```
s1: abc
s2: abc
s1: avc
s2: abc
```

20.4.1 Η Μέθοδος *assign()*

Η (*std*::)*string* έχει και τη μέθοδο *assign()* που δεν είναι κάτι διαφορετικό από τον `operator=()`. Την παίρνουμε αλλάζοντας μόνον την επικεφαλίδα της `operator=()`.

```
BString& BString::assign( const BString& rhs )
{
   if ( &rhs != this )  // if ( !αυτοεκχώρηση )
   {
   // Πάρε τη μνήμη που χρειάζεσαι
      char* tmp;
      try {  tmp = new char[rhs.bsReserved];  }
      catch( bad_alloc& )
```

```
      {  throw BStringXptn( "assign", BStringXptn::allocFailed );  }
   // "Καθάρισε" την παλιά τιμή
      delete[] bsData;
   // Αντίγραψε την τιμή του rhs
      bsData = tmp;
      bsReserved = rhs.bsReserved;
      for ( int k(0); k < rhs.bsLen; ++k )
         bsData[k] = rhs.bsData[k];
      bsLen = rhs.bsLen;
   }
   return *this;
} // BString::assign
```

Αυτή η μορφή είναι καλή για λόγους επίδειξης αλλά έχουμε το ίδιο πρόβλημα που είχαμε με τις *length()* και *size()*: τη συνωνυμία και η εγγύηση για το ότι *assign()* και `operator=()` κάνουν ακριβώς την ίδια δουλειά είναι ο **inline** ορισμός:

```
class BString
{
public:
// . . .
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs ) { return (*this = rhs); }
// . . .
private:
// . . .
}; // BString
```

και όχι το "copy/paste". Γιατί; Διότι πέρα από το αρχικό γράψιμο υπάρχουν και οι διορθώσεις, οι προσαρμογές και τα παρόμοια που μόνο η δεύτερη μορφή εγγυάται ότι θα γίνουν και στις δύο μορφές αυτομάτως. Και δεν θα «γίνει μπέρδεμα» αν πέσει καμιά εξαίρεση που δίνει για προέλευση "`operator=`" ενώ το πρόγραμμα καλούσε την *assign()*; Ε, μη πνίγεσαι σε μια κουταλιά νερό...

20.4.2 Και Μια Εκχώρηση που *δεν* Ορίσαμε

Αν έχουμε δηλώσει:

```
   std:string s0;
```

επιτρέπεται να δώσουμε:

```
   s0 = "what\'s this\?";  cout << "    s0: " << s0 << endl;
```

και να πάρουμε:

```
    s0: what's this?
```

Για να αποκτήσουμε αυτήν τη δυνατότητα και στον *BString* θα πρέπει να (ξανα)επιφορτώσουμε τον "`=`" με μια:

```
   BString& operator=( const char* rhs );
```

Δεν είναι απαραίτητο· δοκίμασε τις

```
   BString s2;

   s2 = "what\'s this\?";  cout << "s2: " << s2.c_str() << endl;
```

Θα πάρεις:

```
s2: what's this?
```

Πώς το καταφέραμε αυτό; Η εκχώρηση, πριν κάνει αυτά που περιγράφουμε στην επιφόρτωση του "`=`", κάνει και κάτι άλλο: «*η τιμή* (της παράστασης που έχουμε δεξιά) *μετατρέπεται στον τύπο της μεταβλητής* (που έχουμε αριστερά)» αν αυτό είναι εφικτό (§2.2, §11.3). Δηλαδή, εκτελείται η:

```
   s2 = static_cast<BString>( "what\'s this\?" );
```

ή, στην περίπτωσή μας:

```
   s2 = BString( "what\'s this\?" );
```

Η μετατροπή τιμής "`const char*`" σε τιμή *BString* είναι εφικτή και γίνεται με τον ερήμην («2 σε 1») δημιουργό της κλάσης. Για να πεισθείς άλλαξε (προσωρινώς) τον δημιουργό ως εξής:

```
BString::BString( const char* rhs )
{
cout << "In default constructor; rhs: " << rhs << endl;
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

   try {  bsData = new char [bsReserved];  }
   catch( bad_alloc )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BString
```

και θα πάρεις:

```
In default constructor; rhs:
In default constructor; rhs: what's this?
s2: what's this?
```

Η πρώτη γραμμή προέρχεται από τη δήλωση της *s2* και η δεύτερη από τη μετατροπή.

20.5 Το Βέλος "`this`"

Στην προηγούμενη παράγραφο χρειαστήκαμε, μέσα σε μια μέθοδο, βέλος προς το αντικείμενο-ιδιοκτήτη της μεθόδου. Πώς το βρίσκουμε αυτό; Ένας τρόπος θα ήταν να πάρουμε ένα βέλος προς το πρώτο μέλος του αντικειμένου, αφού, όπως ξέρουμε από τις δομές, τα δύο βέλη είναι ίσα. Φυσικά, για να μπορέσουμε να το χρησιμοποιήσουμε θα πρέπει να κάνουμε την κατάλληλη (ερμηνευτική) τυποθεώρηση, π.χ.:

```
reinterpret_cast<BString*>( &bsData )
```

Κάτι τέτοιο δεν είναι και τόσο κομψό, είναι δύσχρηστο και *κανείς δεν σου εγγυάται ότι ισχύει* (§15.6)· ας το έχεις δοκιμάσει σε πολλούς μεταγλωττιστές.

Η C++ μας δίνει ένα σίγουρο εργαλείο. Στον ορισμό οποιασδήποτε κλάσης μπορεί να χρησιμοποιηθεί το βέλος `this` (Σχ. 20-4).

♦ ***Για οποιοδήποτε αντικείμενο μιας κλάσης το `this` είναι βέλος προς αυτό το αντικείμενο.***

Ας πούμε, για παράδειγμα, ότι έχουμε ορίσει μια κλάση:

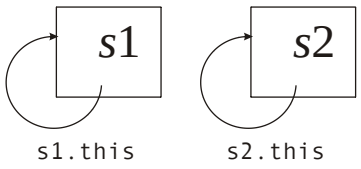
```
class AClass
{
public:
// . . .
   void printthis()
   { cout << this; }
// . . .
private:
// . . .
};
```

και δηλώνουμε:

```
   AClass a, b;
```

Στη συνέχεια δίνουμε τις εντολές:

```
   a.printthis();  cout << "  " << &a << endl;
   b.printthis();  cout << "  " << &b << endl;
```



Σχ. 20-4 Τα βέλη `this` για δύο αντικείμενα. Το βέλος `this` χρησιμοποιείται στις μεθόδους για να δείχνει το ίδιο το αντικείμενο. **Προσοχή!** Τα `s1.this` και `s2.this` είναι μόνο για να δείξουμε την «ιδιοκτησία» του βέλους από το αντικείμενο στο σχήμα και όχι για να το γράψεις στο πρόγραμμά σου.

Δηλαδή, ζητούμε να δούμε τι δείχνει το βέλος `this` για το *a* και τη διεύθυνση του *a*. Τα ίδια ζητούμε και για το *b*. Απότέλεσμα:

```
0x0064fdfc  0x0064fdfc
0x0064fdf4  0x0064fdf4
```

που σημαίνει ότι για το *a* το `this` δείχνει το ίδιο το *a*. Παρόμοια ισχύουν και για το *b*.

20.6 Επιστρέφουμε Τύπο Αναφοράς;

Ο τελεστής εκχώρησης ("`=`") και η *at()* έχουν ένα κοινό χαρακτηριστικό: επιστρέφουν τύπο αναφοράς:

- Ο "`=`" επιστρέφει αναφορά σε ολόκληρο το αντικείμενο.
- Η *at()* επιστρέφει *αναφορά σε μια συνιστώσα του αντικειμένου*.

  Τέτοιες μέθοδοι γενικώς θεωρούνται επικίνδυνες. Γιατί;
- Παράδειγμα: Αν στο πρόγραμμά σου αντί για "`x = y = z`" ή "`x = (y = z)`" γράψεις "`(x = y) = z`" θα αλλάξει η τιμή του *x* αλλά όχι του *y* (όπως θα ήθελες). Αυτό είναι ένα δύσκολο προγραμματιστικό λάθος που μπορεί να σου πάρει αρκετό χρόνο μέχρι να το βρεις. Πάντως, σε κάθε περίπτωση, αυτό που αλλάζει είναι η τιμή ολόκληρου αντικειμένου με τους κανόνες που έχουμε θέσει για αυτές τις αλλαγές.
- Η *at()* ανοίγει μια «παράπλευρη πόρτα» για να μπεί το πρόγραμμα-πελάτης στο αντικείμενο και να κάνει τροποποιήσεις ανεξέλεγκτα και –πιθανόν– να καταστρέψει το αντικείμενο.

  Φυσικά, αν βάζαμε αναφορές "`const`" τότε γλιτώνουμε από όλα αυτά. Γενικώς λοιπόν:[4]

♦ ***Απόφευγε να γράφεις μεθόδους που επιστρέφουν αναφορές χωρίς "`const`".***

  Εδώ εμείς βάλαμε αυτές τις αναφορές

- για να είμαστε σύμφωνοι με τη «φιλοσοφία της C++ (C)» (για τον "`=`") ή
- για να αντιγράψουμε τη λειτουργία της *std*`::`*string* (για την *at()*). Πάντως, θα πρέπει να παραδεχτούμε ότι η δυνατότητα διαχείρισης της τιμής ενός *BString* με τόσο απλό (και οικείο) τρόπο είναι μεγάλο δέλεαρ για να πάρουμε το ρίσκο.

20.7 Μια Κλάση για Διαδρομές Λεωφορείων

Θα δούμε τώρα άλλο ένα παράδειγμα κλάσης της οποίας κάθε αντικείμενο έχει έναν δυναμικό πίνακα του οποίου τα στοιχεία είναι αντικείμενα μιας άλλης κλάσης. Το πρόβλημα:

> *Τα λεωφορεία κάποιου ΚΤΕΛ εξυπηρετούν το επιβατικό κοινό με βάση συγκεκριμένες διαδρομές. Μια διαδρομή έχει ως χαρακτηριστικά την αρχή, το τέλος και –όπου χρειάζεται– κάποιο ενδιάμεσο σημείο (π.χ. Αθήνα – Σούνιο από Μεσόγεια). Έχει ακόμη έναν κωδικό διαδρομής (φυσικός αριθμός).[5] Σε κάθε διαδρομή υπάρχουν συγκεκριμένες στάσεις. Για κάθε μια από αυτές κρατούμε την απόσταση από την αφετηρία σε km και το κόμιστρο από την αφετηρία σε €. Δεν υπάρχουν στάσεις με το ίδιο όνομα. Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία. Σε μια διαδρομή θα πρέπει να έχουμε δυνατότητα να προσθέσουμε νέα σταση και να διαγράψουμε στάση που*

---

[4] Ο *Κανόνας OBJ35* του (CERT 2009) λέει: «*Do not return references to private data*» με σκεπτικό: Αν μια μέθοδος κλάσης επιστρέφει αναφορά ή βέλος προς εσωτερικά δεδομένα, αυτά μπορεί να αλλαχτούν από μη αξιόπιστο κώδικα.

[5] Τα *αρχή*, *τέλος* και *ενδιάμεσο* απαρτίζουν το κλειδί μιας διαδρομής. Ο κωδικός είναι ένα *υποκατάστατο* κλειδί (§15.5.1).

υπάρχει. Θέλουμε μια κλάση, ας την πούμε Route, που κάθε της αντικείμενο θα περιγράφει μια τέτοια διαδρομή.

Στο αρχείο kvadra.txt υπάρχουν τα στοιχεία μιας διαδρομής γραμμένα ως εξής (με το `'\t'` *παριστάνουμε τον οριζόντιο στηλοθέτη (tab)):*

```
102
ΚΑΒΑΛΑ\tΔΡΑΜΑ\t
1Η ΑΓ. ΑΘΑΝΑΣΙΟΥ\t22.1\t3.2
2Η ΑΓ. ΑΘΑΝΑΣΙΟΥ\t22.6\t3.2
 . . .
ΔΙΟΙΚΗΤΗΡΙΟ\t34.5\t5
1Η ΔΟΞΑΤΟΥ\t26.7\t4
 . . .
```

Γράψε πρόγραμμα που θα διαβάζει το αρχείο και θα αποθηκεύει το περιεχόμενό του σε ένα αντικείμενο κλάσης Route. Στη συνέχεια:

1. Θα εισάγει μια νέα στάση με όνομα «`ΜΑΡΜΑΡΑ`*», απόσταση από την αφετηρία 25.5 km και κόμιστρο € 3.7.*

2. Θα αλλάζει σε «`ΔΙΑΣΤΑΥΡΩΣΗ`*» το όνομα της στάσης «*`ΣΤΑΥΡΟΣ`*» –αν υπάρχει.*

3. Θα φυλάγει το τροποιημένο δρομολόγιο στο αρχείο kvadranew.txt στην ίδια μορφή που ήταν και το αρχικό.

20.7.1 Η Κλάση για τις Στάσεις

Το πρώτο που θα κάνουμε είναι να ορίσουμε μια κλάση για τις στάσεις. Θα είναι κάπως έτσι:

```
class RouteStop
{
public:
// . . .
private:
   string sName;  // όνομα στάσης
   float  sDist;  // απόσταση από αφετηρία σε km
   float  sFare;  // τιμή εισιτηρίου από την αφετηρία
}; // RouteStop
```

Γιατί βάλαμε "`class`" και όχι "`struct`"; Διότι θα πρέπει να έχουμε

$$sDist \geq 0 \wedge sFare \geq 0$$

Έχουμε δηλαδή *μη τετριμμένη αναλλοίωτη*.

Κατ' αρχάς να ορίσουμε έναν δημιουργό. Δηλώνουμε:

```
   RouteStop( string aName="", float aDist=0, float aFare=0 );
```

και ορίζουμε:

```
RouteStop::RouteStop( string aName, float aDist, float aFare )
{
   if ( aName.empty() && (aDist > 0 || aFare > 0) )
      throw RouteStopXptn( "RouteStop", RouteStopXptn::noName );
   if ( aDist < 0 )
      throw RouteStopXptn( "RouteStop",
                           RouteStopXptn::negDist, aDist );
   if ( aFare < 0 )
      throw RouteStopXptn( "RouteStop",
                           RouteStopXptn::negFare, aFare );
   sName = aName;
   sDist = aDist;
   sFare = aFare;
} // RouteStop::RouteStop
```

Πρόσεξε τον πρώτο έλεγχο: Δεν επιτρέπεται να δηλώσεις στάση με απόσταση ή/και κόμιστρο χωρίς να δώσεις όνομα στάσης. Πάντως η κατάσταση:

$$sName.empty() \wedge sDist == 0 \wedge sFare == 0$$

είναι δεκτή! Είναι η κατάσταση που βρίσκεται το *oneStop* μετά τη δήλωση:

```
   RouteStop oneStop;
```

Εκείνο το «*Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία*» δεν θα το βάλουμε στην αναλλοίωτη; Πρόσεξε: αυτό έχει να κάνει όχι με μια στάση αλλά με ένα σύνολο στάσεων· άρα *αυτό έχει να κάνει με την κλάση διαδρομών και όχι με την κλάση των στάσεων*.

Στη συνέχεια θα εξοπλίσουμε την κλάση μας και με άλλες μεθόδους που θα χρειαζόμαστε για τη λύση του προβλήματός μας.

Η κλάση εξαιρέσεων για τη *RouteStop* είναι (προς το παρόν):

```
struct RouteStopXptn
{
   enum { noName, negDist, negFare };
   char         funcName[100];
   unsigned int errCode;
   float        errVal;
   RouteStopXptn( const char* fn, int ec, float ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal = ev;  }
}; // RouteStopXptn
```

Σημείωση:▶

Τώρα είδαμε πώς είναι ο δημιουργός αντιγραφής και ο τελεστής εκχώρησης της *BString* και γιατί μπορεί –κατ' αρχήν– να ρίξουν εξαίρεση *BStringXptn* με κωδικό *allocFailed*. Έτσι, καταλαβαίνεις αυτό που λέγαμε στην §Prj03.7: οι αντίστοιχες συναρτήσεις της *string* μπορεί –κατ' αρχήν– να ρίξουν *std*::*bad_alloc*.

Και ερχόμενοι στο προκείμενο, να καταλάβουμε ότι και οι αντίστοιχες συναρτήσεις της *RouteStop* –που έχει το "`string sName`"– μπορεί να ρίξουν (*std*::) *bad_alloc*. Δεν πρέπει να κάνουμε κάτι; Πρέπει (κατ' αρχήν)! Αλλά δεν θα κάνουμε! ◀

20.7.2 Η Κλάση για τις Διαδρομές

Ας πάμε τώρα να γράψουμε την κλάση *Route* με βάση την περιγραφή του προβλήματος:

```
class Route
{
public:
// . . .
private:
   unsigned int rCode;      // κωδικός διαδρομής
   string       rFrom;      // αφετηρία
   string       rTo;        // τέρμα
   string       rInBetween; // ενδιάμεσος
   RouteStop*   rAllStops;  // πίνακας στάσεων διαδρομής
   unsigned int rNoOfStops; // πλήθος στάσεων
}; // Route
```

Με το μέλος *rAllStops* θα χειριζόμαστε έναν δυναμικό πίνακα. Θα χρησιμοποιήσουμε την τεχνική που μάθαμε στη *BString*: Δηλώνουμε άλλο ένα μέλος,

```
   unsigned int rReserved;  // πλήθος στοιχείων που παραχωρήθηκαν
```

και πρέπει να αποφασίσουμε για το «κβάντο» αύξησης.

Μια διαδρομή έχει συνήθως μερικές δεκάδες στάσεις, κάτι σαν 10, 20, 30. Το 5 είναι μια λογική τιμή για το κβάντο:

```
   enum {  rIncr = 5  };   // το "κβάντο" αύξησης δυναμικής μνήμης
                           // για τον πίνακα rAllStops
```

20.7.2.1 Η Αναλλοίωτη

Ποια είναι η αναλλοίωτη της κλάσης μας; Εκτός από την $rCode \geq 0$ και τη συνθήκη διαχείρισης του δυναμικού πίνακα:

$$(0 \leq rNoOfStops < rReserved) \;\&\&\; (rReserved \;\%\; rIncr == 0)$$

το ουσιώδες μέρος είναι αυτό: «*Δεν υπάρχουν στάσεις με το ίδιο όνομα. Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία.*» Ακόμη, υπάρχει και κάτι άλλο που δεν γράφεται στην περιγραφή του προβλήματος ως αυτονόητο: *δεν υπάρχουν δύο στάσεις στην ίδια απόσταση από την αφετηρία*. Τα παραπάνω διατυπώνονται συμβολικώς ως εξής:

$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sName \neq \text{rAllStops}[k].sName)\;\&\&$$
$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)\;\&\&$$
$$(\forall\, j, k: 0\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[j].sDist > \text{rAllStops}[k].sDist) \Rightarrow$$
$$(\text{rAllStops}[j].sFare \geq \text{rAllStops}[k].sFare))$$

Εδώ πρέπει να λύσουμε ένα πρόβλημα: Αν πρέπει να προσθέσουμε μια νέα στάση πώς ελέγχουμε τη συμμόρφωσή της με την αναλλοίωτη; Θα πρέπει

- να βρούμε τη στάση με τη μέγιστη απόσταση από την αφετηρία που δεν είναι μεγαλύτερη από την απόσταση της νέας στάσης (αν υπάρχει),
- να βρούμε τη στάση με τη ελάχιστη απόσταση από την αφετηρία που δεν είναι μικρότερη από την απόσταση της νέας στάσης (αν υπάρχει),
- να ελέγξουμε αν η ισχύει η συνθήκη για τις τρεις αυτές στάσεις.

Αυτά μπορεί να γίνουν πολύ εύκολα αν έχουμε τον πίνακα ταξινομημένο ως προς την απόσταση από την αφετηρία:

$$\forall\, k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{-}1].sDist < \text{rAllStops}[k].sDist$$

Αν πρέπει να εισαγάγουμε την *aStop* και βρούμε ότι

$$\text{rAllStops}[k\text{-}1].sDist < aStop.sDist < \text{rAllStops}[k].sDist$$

η εισαγωγή πρέπει να γίνει στη θέση k και η αναλλοίωτη μας λέει ότι θα πρέπει να ισχύει και η:

$$\text{rAllStops}[k\text{-}1].sFare \leq aStop.sFare \leq \text{rAllStops}[k].sFare$$

Εδώ τώρα πρόσεξε ένα πρόβλημα:

- Αν το k έχει τιμή 0 (η εισαγωγή πρέπει να γίνει *πριν από το πρώτο* στοιχείο) πως θα ελέγξουμε την ανισότητα στα αριστερά;
- Αν το k έχει τιμή *rNoOfStops* (η εισαγωγή πρέπει να γίνει *μετά το τελευταίο* στοιχείο) πώς θα ελέγξουμε την ανισότητα στα δεξιά;

Μια λύση είναι να βάλουμε ξεχωριστούς ελέγχους για εισαγωγή στην πρώτη και στην τελευταία θέση. Μια άλλη λύση είναι οι φρουροί:

- Βάζουμε τα στοιχεία του πίνακα στις θέσεις $1\,..\,rNoOfStops$.
- Στη θέση 0 βάζουμε φρουρό

$$\text{rAllStops}[0].sDist == 0 \;\&\&\; \text{rAllStops}[0].sFare == 0$$

- Στη θέση *rNoOfStops*+1 βάζουμε φρουρό:

$$\text{rAllStops}[rNoOfStops\text{+}1].sDist == FLT\_MAX \;\&\&$$
$$\text{rAllStops}[rNoOfStops\text{+}1].sFare == FLT\_MAX$$

Αυτά μας αλλάζουν και τη σχέση μεταξύ *rNoOfStops* και *rReserved*:

$$(0 \leq rNoOfStops \leq rReserved\text{-}2) \;\&\&\; (rReserved \;\%\; rIncr == 0)$$

Τελικώς, μπορούμε να πούμε ότι η αναλλοίωτη της κλάσης είναι:

$$I \equiv I_D \;\&\&\; I_I$$

όπου (προσοχή στις αλλαγές των δεικτών!)

$$I_D \equiv (rCode \geq 0)\;\&\&$$
$$(\forall\, j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sName \neq \text{rAllStops}[k].sName)\;\&\&$$
$$(\forall\, j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)\;\&\&$$

$(\forall\ j, k: 1\,..\,rNoOfStops \bullet (\text{rAllStops}[j].sDist > \text{rAllStops}[k].sDist) \Rightarrow$
$(\text{rAllStops}[j].sFare \geq \text{rAllStops}[k].sFare))$

είναι το κομμάτι που εξαρτάται από τους **επιχειρησιακούς κανόνες** (business rules) που έχουν σχέση με το πρόβλημα:

- *Ο κωδικός είναι θετικός ακέραιος* (το "0" το προσθέτουμε εμείς και μόνο για δήλώσεις όπως: "`Route oneStop`").
- *Το κόμιστρο από την αφετηρία είναι αύξουσα συνάρτηση της απόστασης από την αφετηρία.*
- Οι στάσεις της διαδρομής είναι ένα *σύνολο*. Τα στοιχεία του συνόλου διαφέρουν ανά δύο ως προς το όνομα (*sName*) και ως προς την απόσταση από την αφετηρία (*sDist*).

$$I_I \equiv (0 \leq rNoOfStops \leq rReserved\text{-}2)\ \&\&\ (rReserved\ \%\ rIncr == 0)\ \&\&$$
$$(\forall\ k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{+}1].sDist > \text{rAllStops}[k].sDist)\ \&\&$$
$$(\text{rAllStops}[0].sDist == 0\ \&\&\ \text{rAllStops}[0].sFare == 0)\ \&\&$$
$$(\text{rAllStops}[rNoOfStops\text{+}1].sDist == FLT\_MAX\ \&\&$$
$$\text{rAllStops}[rNoOfStops\text{+}1].sFare == FLT\_MAX)$$

είναι το κομμάτι που εξαρτάται από τις επιλογές που κάναμε για την υλοποίηση. Αυτό το κομμάτι θα αλλάξει αν εγκαταλείψουμε τον δυναμικό πίνακα και επιλέξουμε άλλη δομή δεδομένων για την παράσταση των στάσεων. Σχετικώς να επισημάνουμε και το εξής: Αν ισχύει το τμήμα της I_I

$$(\forall\ k: 1\,..\,rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k\text{+}1].sDist > \text{rAllStops}[k].sDist)$$

προφανώς ισχύει και το τμήμα της I_D

$$(\forall\ j, k: 1\,..\,rNoOfStops \bullet (j \neq k \Rightarrow \text{rAllStops}[j].sDist \neq \text{rAllStops}[k].sDist)$$

αλλά αυτό δεν είναι λόγος να διαγράψουμε τη δεύτερη αφού αυτή ισχύει ακόμη και αν αλλάξουμε την υλοποίηση που επιλέγουμε για τις στάσεις.

20.7.2.2 Ένας Δημιουργός

Το πρώτο που θα κάνουμε είναι να γράψουμε έναν ερήμην δημιουργό. Τον δηλώνουμε:

```
   Route( int rc=0 );
```

και τον ορίζουμε ως εξής:

```
Route::Route( int rc )
{
   if ( rc < 0 )
      throw RouteXptn( "Route", RouteXptn::negCode, rc );
   rCode = rc;
   try
   {
      rReserved = rIncr;
      rAllStops = new RouteStop[ rReserved ];
      rNoOfStops = 0;
      rAllStops[0] = RouteStop( "guard0", 0.0, 0.0 );                  // φρουροί
      rAllStops[rNoOfStops+1] = RouteStop( "guardPte", FLT_MAX, FLT_MAX );
      rFrom = "";
      rTo = "";
      rInBetween = "";
   }
   catch( bad_alloc& )
   {
      throw RouteXptn( "Route", RouteXptn::allocFailed );
   }
} // Route::Route
```

Όπως βλέπεις, κατά βάση κάνουμε αυτά που ξέρουμε από τη *BString* και δεν αλλάζουν πολύ από το ότι εδώ έχουμε πίνακα αντικειμένων και όχι τιμών τύπου `char`. Η σημαντική διαφορά είναι οι δύο φρουροί.

20.7.2.3 Μέθοδοι

Θα χρειαστούμε μεθόδους για να δώσουμε τιμές στα μέλη *rFrom*, *rTo* και *rInBetween* (ή να αλλάξουμε τις τιμές τους). Είναι πολύ απλές και θα τις ορίσουμε **inline**:

```
   void setFrom( string aName )  {  rFrom = aName;  }
   void setTo( string aName )  {  rTo = aName;  }
   void setInBetween( string aName )  {  rInBetween = aName;  }
```

Πριν προχωρήσουμε στον χειρισμό των στοιχείων του πίνακα να πούμε ότι σε τέτοιες περιπτώσεις είναι χρήσιμη μια μέθοδος που «καθαρίζει» τον πίνακα ώστε το ίδιο αντικείμενο να ετοιμαστεί για να φιλοξενήσει άλλη διαδρομή:

```
void Route::clearRouteStops()
{  rAllStops[1] = rAllStops[rNoOfStops+1];
   rNoOfStops = 0;  }
```

Όπως βλέπεις δεν αναλυκλώνει τη μνήμη που έχει το αντικείμενο και επομενως δεν αλλάζει την τιμή του *rReserved*.

20.7.2.4 Χειρισμός Στοιχείων Πίνακα

Διαγραφή: Κατ' αρχάς ας ανταποκριθούμε στην απαίτηση «*να διαγράψουμε στάση που υπάρχει*», μια και η διαγραφή που είναι πιο απλή από την εισαγωγή στάσης.[6] Θα γράψουμε μια μέθοδο με όνομα *deleteRouteStop()* και προδιαγραφές:

```
// I
   aRoute.deleteRouteStop( stopName );
// I && (η διαδρομή aRoute δεν έχει στάση με όνομα stopName)
```

Αν δεν είχαμε την απαίτηση της ταξινόμησης τα πράγματα θα ήταν απλά:

```
void Route::deleteRouteStop( string stopName )
{
   if ( υπάρχει στάση με όνομα stopName στη θέση ndx )
   {
      αντίγραψε στη θέση ndx το τελευταίο στοιχείο του πίνακα
      --rNoOfStops;
   }
}
```

Τώρα, τα πράγματα είναι πιο πολύπλοκα:

```
   if ( υπάρχει στάση με όνομα stopName στη θέση ndx )
   {
      μετακίνησε τα στοιχεία
         rAllStops[ndx+1] ... rAllStops[rNoOfStops+1]
      στα
         rAllStops[ndx] ... rAllStops[rNoOfStops];
      --rNoOfStops;
   }
```

Αν γράψουμε μια

```
int Route::findNdx( const string& aName ) const
{
   rAllStops[rNoOfStops+1].setName( aName );
   int ndx( 1 );
   while ( rAllStops[ndx].getName() != aName )  ++ndx;
   if ( ndx > rNoOfStops ) ndx = -1;
   return ndx;
} // Route::findNdx
```

–που μας επιστρέφει τον δείκτη του στοιχείου που έχει στο *sName* τιμή *aName* ή –1 αν δεν βρει τέτοιο στοιχείο– μπορούμε να υλοποιήσουμε τη *deleteRouteStop()* ως εξής:

---

[6] Με όρους θεωρίας συνόλων –αν ονομάσουμε *A* το σύνολο στάσεων και *x* τη στάση που θέλουμε να διαγράψουμε– η πράξη που θέλουμε να υλοποιήσουμε είναι η: $A = A \setminus \{ x \}$ (το "=" με την έννοια της εκχώρησης).

```
void Route::deleteRouteStop( string stopName )
{
   int ndx( findNdx(stopName) );
   if ( ndx >= 0 )
   {
      try {  erase1RouteStop( ndx );  }
      catch( bad_alloc& )
      {  throw RouteXptn( "deleteRouteStop",
                          RouteXptn::allocFailed );  }
   }
} // Route::deleteRouteStop
```

όπου

```
void Route::erase1RouteStop( int ndx )
{
   for ( int k(ndx); k <= rNoOfStops; ++k )
      rAllStops[k] = rAllStops[k+1];
   --rNoOfStops;
} // Route::erase1RouteStop
```

Η δεύτερη συνάρτηση «σβήνει» το στοιχείο που βρίσκεται στη θέση *ndx* σε έναν ταξινομημένο πίνακα. Η πρώτη παίρνει την απόφαση για το αν θα πρέπει να σβηστεί κάποιο στοιχείο. Κάθε πρόγραμμα-πελάτης της κλάσης θα έχει πρόσβαση στην πρώτη συνάρτηση αλλά όχι στη δεύτερη που θα δηλωθεί σε περιοχή **private**.

Και εκείνα τα **try**/**catch** γιατί τα βάλαμε; Για όλα εκείνα τα «κατ' αρχήν» που λέγαμε πιο πάνω: κάποια από τις "`rAllStops[k] = rAllStops[k+1]`" της *erase1RouteStop()* μπορεί να ρίξει *bad_alloc*!

Η *deleteRouteStop()* είναι σύμφωνη με τις προδιαγραφές που βάλαμε αλλά μπορεί να σου φαίνεται περίεργο το ότι δεν αντιδρά στην περίπτωση που δεν υπάρχει στάση με το όνομα που μας ενδιαφέρει. Η αλήθεια είναι ότι σε μερικές περιπτώσεις το πρόγραμμα-πελάτης θα θέλει να το ξέρει. Για να λύσουμε αυτό το πρόβλημα θα εφοδιάσουμε την κλάση μας με την εξής μέθοδο:[7]

```
bool Route::find1RouteStop( string stopName )
{
   return ( findNdx(stopName) >= 0 );
} // Route::find1RouteStop
```

Και γιατί να μην χρησιμοποιήσουμε τη *findNdx()*; Η *findNdx()* βγάζει ως αποτέλεσμα τον δείκτη του στοιχείου στον πίνακα. Αυτός όμως έχει σχέση με την εσωτερική παράσταση του συνόλου των στάσεων. Αν τη δηλώσουμε "`public`" παραβιάζουμε την *απόκρυψη πληροφορίας* (§19.3.1). Αν αργότερα αλλάξουμε την εσωτερική παράσταση –π.χ. σε δένδρο– θα πρέπει να διορθώσουμε όλες τις εφαρμογές που χρησιμοποιούν την κλάση και τη *findNdx()*. Για τον λόγο αυτόν:

- Δηλώνουμε "`public`" τη *find1RouteStop()*.
- Δηλώνουμε "`private`" τη *findNdx()* που είναι *κρυμμένη μέθοδος* και όχι βοηθητική συνάρτηση (π.χ. σαν τη *lastDay()*).

Παρατηρήσεις: ▶

Πριν προχωρήσουμε θα κάνουμε δύο παρατηρήσεις σχετικώς με τη *findNdx()*.

1. Η *findNdx()* θέτει απαιτήσεις για την κλάση *RouteStop*: Πρέπει να έχει μεθόδους *setName()* και *getName()*.

2. Μιλούσαμε στην αρχή αυτού του κεφαλαίου για μηνύματα «*που μπορεί να δέχεται, ένα αντικείμενο ... είτε από άλλα αντικείμενα που "ζούν" στο [ίδιο] πρόγραμμα*». Εδώ δώσαμε τη δυνατότητα σε ένα αντικείμενο *aRoute* κλάσης *Route* να στέλνει

- Στο αντικείμενο –κλάσης *RouteStop*– *aRoute*`.rAllStops[`*rNoOfStops*+1`]` μήνυμα *setName()*.

[7] Με όρους θεωρίας συνόλων η *find1RouteStop()* υλοποιεί την $x \in A$.

- Σε ένα ή περισσότερα στοιχεία του πίνακα *aRoute*.rAllStops –που είναι αντικείμενα κλάσης *RouteStop*– μήνυμα *getName()*.◀

Εισαγωγή: Για να ανταποκριθούμε στην απαίτηση «*να προσθέσουμε νέα σταση*», θα γράψουμε μια μέθοδο με όνομα *addRouteStop()* και προδιαγραφές:

```
// I
   aRoute.addRouteStop( aStop );
// I && (η διαδρομή aRoute έχει τη στάση aStop)
```

Για να κάνουμε εισαγωγή της νέας στάσης θα πρέπει:[8]

- Να μην υπάρχει άλλη στάση με το ίδιο όνομα (*aStop*.*sName*) και φυσικά
- να μην υπάρχει άλλη σταση rAllStops[*ndx*] στην ίδια απόσταση από την αφετηρία (*aStop*.*sDist*). «Ίδια»; Τι θα πει ίδια; Προς το παρόν ας πούμε ότι σημαίνει «ίση» με την έννοια *aStop*.*sDist* == rAllStops[*ndx*].*sDist*.

Ας πούμε λοιπόν ότι δίνουμε:

```
   int nmNdx( findNdx(aStop.getName()) );
```

Τι θα κάνουμε αν πάρουμε *nmNdx* ≥ 1, αν δηλαδή βρούμε στάση με το ίδιο όνομα; Υπάρχουν δύο περιπτώσεις:

- Αν έχουμε επιπλέον *aStop*.*sDist* == rAllStops[*nmNdx*].*sDist* και *aStop*.*sFare* == rAllStops[*nmNdx*].*sFare* τότε «*η διαδρομή aRoute έχει τη στάση aStop*» χωρίς να κάνουμε οτιδήποτε! (θυμίσου: δεν κάναμε οτιδήποτε και στην περίπτωση της διαγραφής, όταν δεν βρήκαμε στάση με το ίδιο όνομα.)
- Αν όμως έχουμε διαφορές, δηλαδή *aStop*.*sDist* != rAllStops[*nmNdx*].*sDist* είτε *aStop*.*sFare* != rAllStops[*nmNdx*].*sFare* τότε έχουμε πρόβλημα και θα πρέπει να ρίξουμε εξαίρεση.

Δηλαδή:

```
   int nmNdx( findNdx(aStop.getName()) );
   if ( nmNdx > 0 )  // name found
   {
      if ( rAllStops[nmNdx].getDist() != aStop.getDist() ||
           rAllStops[nmNdx].getFare() != aStop.getFare() )
         throw RouteXptn( "addRouteStop",
                          RouteXptn::invalidArgument,
                          rAllStops[nmNdx], aStop );
   }
```

Ας πούμε τώρα ότι δεν έχουμε βρει στάση με το ίδιο όνομα· ψάχνουμε να βρούμε στάση στην ίδια απόσταση (*aStop*.*sDist*) από την αφετηρία. Αν υπάρχει θα έχει διαφορετικό όνομα· θα πρέπει λοιπόν να ρίξουμε εξαίρεση. Αφού έχουμε πίνακα ταξινομημένο στο *sDist* και φρουρούς στην αρχή και στο τέλος μπορούμε να χρησιμοποιήσουμε τη *binSearch()* της §9.6 αφού πρώτα

- τη μετατρέψουμε σε περίγραμμα

```
template < class T >
unsigned int binSearch( const T v[], int last, const T& x )
```

- επιφορτώσουμε τον τελεστή "<" για τη *RouteStop* ως εξής:

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < 0 );
} // operator<( const RouteStop
```

Αν δεν υπάρχει στάση στην ίδια απόσταση η *binSearch()* θα μας επιστρέψει τη θέση στην οποία θα πρέπει να εισαχθεί η νέα στάση ώστε ο πίνακας να είναι ταξινομημένος:

```
   else  // name not found
   {
```

[8] Με όρους θεωρίας συνόλων η *insert1RouteStop()* υλοποιεί την: $A = A \cup \{ x \}$.

```
      int distNdx( binSearch(rAllStops, rNoOfStops, aStop) );
      if ( rAllStops[distNdx].getDist() == aStop.getDist() )
         throw RouteXptn( "addRouteStop", RouteXptn::diffName,
                          rAllStops[distNdx], aStop );
```

αλλιώς θα πρέπει να έχουμε:

rAllStops[*distNdx*-1].*getDist*() < *aStop*.*getDist*() ≤ rAllStops[*distNdx*].*getDist*()

Πριν κάνουμε την εισαγωγή θα πρέπει να ελέγξουμε και τον περιορισμό για το κόμιστρο. Η αναλλοίωτη μας λέει ότι θα πρέπει να ισχύει και η:

rAllStops[*distNdx*-1].*getFare*()≤ *aStop*.*getFare*() ≤ rAllStops[*distNdx*].*getFare*()

```
      if ( rAllStops[distNdx-1].getFare() > aStop.getFare() )
         throw RouteXptn( "addRouteStop", RouteXptn::fareErr,
                          rAllStops[distNdx-1], aStop );
      if ( aStop.getFare() > rAllStops[distNdx].getFare() )
         throw RouteXptn( "addRouteStop", RouteXptn::fareErr,
                          aStop, rAllStops[distNdx] );
```

Αν περάσουμε και αυτούς τους ελέγχους είμαστε σχεδόν έτοιμοι για την εισαγωγή. Θα πρέπει μόνο να σιγουρέψουμε ότι έχουμε την απαραίτητη μνήμη:

```
      if ( rReserved <= rNoOfStops+2 )
      {
         try {  renew( rAllStops, rNoOfStops+2, rReserved+rIncr );
                rReserved += rIncr;  }
         catch( MyTmpltLibXptn& )
         {  throw RouteXptn( "addRouteStop",
                             RouteXptn::allocFailed );  }
      }
```

Πρόσεξε τη συνθήκη της **if**: Είναι η !*S* όπου

$$S \equiv rReserved > rNoOfStops+2$$

η συνθήκη: «*υπάρχουν διαθέσιμες θέσεις στον πίνακα*».

Πρόσεξε ακόμη ότι στην κλήση της *renew()* ζητούμε να αντιγραφούν στον νέο πίνακα τα πρώτα *rNoOfStops*+2 στοιχεία του παλιού.

Μετά από αυτό μπορούμε να κάνουμε την εισαγωγή:

```
      μετακίνησε τα στοιχεία
         rAllStops[distNdx] ... rAllStops[rNoOfStops+1]
      στα
         rAllStops[distNdx+1] ... rAllStops[rNoOfStops+2];
      rAllStops[distNdx] = aStop;
      ++rNoOfStops;
```

Οι μετακινήσεις στοιχείων «ανοίγουν κενό»[9] στη θέση *distNdx*. *Οι μετακινήσεις θα πρέπει να γίνουν από το τέλος προς την αρχή*:

```
      for ( int k(rNoOfStops+1); k >= distNdx; --k )
         rAllStops[k+1] = rAllStops[k];
      rAllStops[distNdx] = aStop;
      ++rNoOfStops;
```

Να ολόκληρη η μέθοδος, επιτέλους!

```
void Route::addRouteStop( const RouteStop& aStop )
{
   int nmNdx( findNdx(aStop.getName()) );
   if ( nmNdx > 0 )  // name found
   {
      if ( rAllStops[nmNdx].getDist() != aStop.getDist() ||
           rAllStops[nmNdx].getFare() != aStop.getFare() )
         throw RouteXptn( "addRouteStop",
                          RouteXptn::invalidArgument,
                          rAllStops[nmNdx], aStop );
   }
   else // name not found
```

[9] Φυσικά δεν υπάρχει «κενό»! Έτσι; Τρόπος του λέγειν...

```
   {
      int distNdx( binSearch(rAllStops, rNoOfStops, aStop) );
      if ( rAllStops[distNdx].getDist() == aStop.getDist() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::diffName,
                          rAllStops[distNdx], aStop );
      if ( rAllStops[distNdx-1].getFare() > aStop.getFare() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::fareErr,
                          rAllStops[distNdx-1], aStop );
      if ( aStop.getFare() > rAllStops[distNdx].getFare() )
         throw RouteXptn( "insert1RouteStop", RouteXptn::fareErr,
                          aStop, rAllStops[distNdx] );
      insert1RouteStop( aStop, distNdx );
   }
} // Route::addRouteStop
```

όπου

```
void Route::insert1RouteStop( const RouteStop& aStop,
                              int distNdx )
{
   if ( rReserved <= rNoOfStops+2 )
   {
      try {  renew( rAllStops, rNoOfStops+2, rReserved+rIncr );
             rReserved += rIncr;  }
      catch( MyTmpltLibXptn& )
      {  throw RouteXptn( "insert1RouteStop", RouteXptn::allocFailed );  }
   }
   for ( int k(rNoOfStops+1); k >= distNdx; --k )
   {
      try {  rAllStops[k+1] = rAllStops[k];  }
      catch( bad_alloc& )
      {  throw RouteXptn( "insert1RouteStop", RouteXptn::allocFailed );  }
   }
   rAllStops[distNdx] = aStop;
   ++rNoOfStops;
} // Route::insert1RouteStop
```

Η *insert1RouteStop()*, όπως και η *erase1RouteStop()*, δηλώνεται ως **private**.

Έτσι, η εισαγωγή της στάσης *"MAPMAPA"* θα γίνει ως εξής:

```
      if ( oneRoute.find1RouteStop("MAPMAPA") )
         cout << "Στάση MAPMAPA υπάρχει" << endl;
      else
         oneRoute.addRouteStop( RouteStop("MAPMAPA", 25.5, 3.7) );
```

Τέλος, να σημειώσουμε ότι η υλοποίηση της *addRouteStop()* απαιτεί δύο ακόμη μεθόδους για τη *RouteStop*: τις *getDist()* και *getFare()*.

Τροποποίηση – Ανάκτηση: Στην περιγραφή του προβλήματος υπάρχει η απαίτηση για το πρόγραμμα να «*αλλάζει το όνομα της στάσης "ΣΤΑΥΡΟΣ" –αν υπάρχει– σε "ΔΙΑΣΤΑΥΡΩΣΗ"*.» Μήπως θα πρέπει να γράψουμε μια μέθοδο *modify1RouteStop()* για να αντιμετωπίζουμε τέτοιες απαιτήσεις; Μετά από αυτά που είδαμε για την *addRouteStop()*, ούτε να το σκεφθούμε. Τι κάνουμε λοιπόν;

```
   if ( υπάρχει στάση με όνομα "ΣΤΑΥΡΟΣ" )
   {
      αντίγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ" στο tmp
      βάλε στο tmp όνομα στάσης "ΔΙΑΣΤΑΥΡΩΣΗ"
      διάγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ"
      κάνε εισαγωγή του tmp
   }
```

Μπορούμε να υλοποιήσουμε όλα τα παραπάνω εκτός από την «*αντίγραψε το στοιχείο με όνομα στάσης "ΣΤΑΥΡΟΣ" στο tmp*». Για αυτήν την αντιγραφή θα μας χρειαστεί η

```
const RouteStop& Route::get1RouteStop( string stopName ) const
{
   int ndx( findNdx(stopName) );
   if ( ndx < 0 )
      throw RouteXptn( "get1RouteStop", RouteXptn::notFound,
```

```
                        stopName.c_str() );
   return rAllStops[ndx];
} // Route::get1RouteStop
```

Τώρα μπορούμε να γράψουμε:

```
      if ( !oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ") )
         cout << "Στάση ΣΤΑΥΡΟΣ δεν υπάρχει" << endl;
      else if ( oneRoute.find1RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ") )
         cout << "Στάση ΔΙΑΣΤΑΥΡΩΣΗ υπάρχει" << endl;
      else
      {
         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
```

Πρόσεξε δύο πράγματα:

- Αν βάζαμε τον τύπο της *get1RouteStop* "`RouteStop&`" και όχι "`const RouteStop&`" θα μπορούσαμε να τροποποιήσουμε το στοιχείο του πίνακα επι τόπου με τις συνέπειες που μπορείς να φανταστείς.
- Όταν γίνεται η κλήση `oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ")` έχουμε διασφαλίσει με τη `oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ")` ότι υπάρχει τέτοια στάση. Γενικώς, αν καλέσεις την *get1RouteStop* χωρίς να εξασφαλιστείς προηγουμένως με τη *find1RouteStop* μπορεί να ριχτεί εξαίρεση.

20.7.3 Οι Κλάσεις Τελικώς

Μετά από όσα είδαμε το περιεχόμενο του `Route.h` είναι:

```
#ifndef _ROUTE_H
#define _ROUTE_H

#include <string>
#include <new>

#include "RouteStop.h"

using namespace std;

class Route
{
public:
   Route( int rc=0 );
   ~Route() {  delete[] rAllStops;  }
   void setFrom( string aName )  {  rFrom = aName;  }
   void setTo( string aName )  {  rTo = aName;  }
   void setInBetween( string aName )  {  rInBetween = aName;  }
   bool find1RouteStop( string stopName ) const;
   const RouteStop& get1RouteStop( string stopName ) const;
   void addRouteStop( const RouteStop& aStop );
   void deleteRouteStop( string stopName );
   void writeToText( ostream& tout ) const;
private:
   enum {  rIncr = 5  };    // το "κβάντο" αύξησης δυναμικής
                            // μνήμης για τον πίνακα rAllStops
   unsigned int rCode;      // κωδικός διαδρομής
   string       rFrom;      // αφετηρία
   string       rTo;        // τέρμα
   string       rInBetween; // ενδιάμεσος
   RouteStop*   rAllStops;  // πίνακας στάσεων διαδρομής
   unsigned int rNoOfStops; // πλήθος στάσεων
   unsigned int rReserved;  // πλήθος στοιχείων που παραχωρήθηκαν
```

```
                              // πάντα πολλαπλάσιο του 5
   int findNdx( const string& aName ) const;
   void insert1RouteStop( const RouteStop& aStop, int distNdx );
   void erase1RouteStop( int ndx );
}; // Route

struct RouteXptn
{
   enum { negCode, allocFailed, invalidArgument, notFound,
          diffName, fareErr };
   char         funcName[100];
   unsigned int errCode;
   int          errIntVal;
   RouteStop      errStop1, errStop2;
   char         errStrVal[100];
   RouteXptn( const char* fn, int ec, int ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errIntVal = ev;  }
   RouteXptn( const char* fn, int ec, const char* astr )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      strncpy( errStrVal, astr, 99 );  errStrVal[99] = '\0';  }
   RouteXptn( const char* fn, int ec,
              const RouteStop& aStop1, const RouteStop& aStop2 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;
      errStop1 = aStop1;  errStop2 = aStop2;  }
}; // RouteXptn

#endif // _ROUTE_H
```

Η *writeToText()* ανταποκρίνεται στην τελευταία απαίτηση και είναι:

```
void Route::writeToText( ostream& tout ) const
{
   tout << rCode << endl;
   tout << rFrom << '\t' << rTo << '\t' << rInBetween << endl;
   for ( int k(1); k <= rNoOfStops; ++k )
   {  rAllStops[k].writeToText( tout );  tout << endl;  }
} // Route::writeToText
```

Όπως βλέπεις, θα πρέπει να γράψουμε μια *writeToText()* και για τη *RouteStop*.

Το περιεχόμενο του **RouteStop.h** είναι:

```
#ifndef _ROUTESTOP_H
#define _ROUTESTOP_H

#include <string>
#include <new>

using namespace std;

class RouteStop
{
public:
   RouteStop( string aName="", float aDist=0, float aFare=0 );
   string getName() const {  return sName;  }
   float getDist() const {  return sDist;  }
   float getFare() const {  return sFare;  }
   void setName( string aName );
   void writeToText( ostream& tout ) const;
private:
   string sName;  // όνομα στάσης
   float  sDist;  // απόσταση από αφετηρία σε km
   float  sFare;  // τιμή εισιτηρίου από την αφετηρία
}; // RouteStop

struct RouteStopXptn
{
```

```
   enum { noName, negDist, negFare };
   char         funcName[100];
   unsigned int errCode;
   double       errVal;
   RouteStopXptn( const char* fn, int ec, double ev=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errVal = ev;  }
}; // RouteStopXptn

bool operator<( const RouteStop& lhs, const RouteStop& rhs );
bool operator>=( const RouteStop& lhs, const RouteStop& rhs );

#endif //_ROUTESTOP_H
```

Η *writeToText()* είναι πολύ απλή:

```
void RouteStop::writeToText( ostream& tout ) const
{
   tout << sName << '\t' << sDist << '\t' << sFare;
} // RouteStop::writeToText
```

Πολύ απλός είναι και ο `operator<` (και ο "`>=`" που είναι ο "`!<`")

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < 0 );
} // operator<( const RouteStop

bool operator>=( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( !(lhs < rhs) );
} // operator>=( const RouteStop
```

20.7.4 Και το Πρόγραμμα

Το πρόγραμμα είναι απλούστατο:

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

#include "Route.cpp"

using namespace std;

void readFromText( RouteStop& aStop, istream& tin );

int main()
{
   try
   {
      ifstream tin( "kvadra.txt" );
      string   name, buf;
      double   dist, fare;

      getline( tin, buf, '\n' );

      Route oneRoute( atoi(buf.c_str()) );

      getline( tin, name, '\t' );  oneRoute.setFrom( name );
      getline( tin, name, '\t' );  oneRoute.setTo( name );
      getline( tin, name, '\n' );  oneRoute.setInBetween( name );

      RouteStop oneStop;
      readFromText( oneStop, tin );
      while ( !tin.eof() )
      {
         oneRoute.addRouteStop( oneStop );
         readFromText( oneStop, tin );
```

```
      } // while
      tin.close();

      if ( oneRoute.find1RouteStop("ΜΑΡΜΑΡΑ") )
         cout << "Στάση ΜΑΡΜΑΡΑ υπάρχει" << endl;
      else
         oneRoute.addRouteStop( RouteStop("ΜΑΡΜΑΡΑ", 25.5, 3.7) );

      if ( !oneRoute.find1RouteStop("ΣΤΑΥΡΟΣ") )
         cout << "Στάση ΣΤΑΥΡΟΣ δεν υπάρχει" << endl;
      else if ( oneRoute.find1RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ") )
         cout << "Στάση ΔΙΑΣΤΑΥΡΩΣΗ υπάρχει" << endl;
      else
      {
         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
      ofstream tout( "kvadranew.txt" );
      oneRoute.writeToText( tout );
      tout.close();
   }
   catch( RouteStopXptn& x )
   {
      switch ( x.errCode )
      {
         case RouteStopXptn::noName:
            cout << "bus stop name cannot be empty in "
                 << x.funcName << endl;
            break;
         case RouteStopXptn::negDist:
            cout << "bus stop distance from start (" << x.errVal
                 << ") cannot be negative in " << x.funcName
                 << endl;
            break;
         case RouteStopXptn::negFare:
            cout << "fare from start to bus stop (" << x.errVal
                 << ") cannot be negative in " << x.funcName
                 << endl;
            break;
         default:
            cout << "unexpected RouteStopXptn exception from "
                 << x.funcName << endl;
      }
   }
   catch( RouteXptn& x )
   {
      switch ( x.errCode )
      {
         case RouteXptn::negCode:
            cout << "negative Route code (" << x.errIntVal
                 << ") in " << x.funcName << endl;
            break;
         case RouteXptn::allocFailed:
            cout << "out of memory in " << x.funcName << endl;
            break;
         case RouteXptn::invalidArgument:
         case RouteXptn::diffName:
            cout << "trying to insert bus stop (";
            x.errStop2.writeToText( cout );
            cout << ") over existing (";
            x.errStop1.writeToText( cout );
            cout << endl;
            break;
         case RouteXptn::notFound:
```

```
            cout << "no stop " << x.errStrVal << " requested by "
                 << x.funcName << endl;
            break;
         case RouteXptn::fareErr:
            cout << "inserting (";
            x.errStop2.writeToText( cout );
            cout << ") violates increasing fare principle (";
            x.errStop1.writeToText( cout );
            cout << endl;
            break;
         default:
            cout << "unexpected RouteXptn exception from "
                 << x.funcName << endl;
      }
   }
} // main
```

όπου:

```
void readFromText( RouteStop& aStop, istream& tin )
{
   string name;
   getline( tin, name, '\t' );
   if ( !tin.eof() )
   {
      string buf;
      getline( tin, buf, '\t' );
      float dist( atof(buf.c_str()) );
      getline( tin, buf, '\n' );
      float fare( atof(buf.c_str()) );
      aStop = RouteStop( name, dist, fare );
   }
} // readFromText
```

Σχόλια; Στη συνέχεια...

20.7.5 Σχόλια, Παρατηρήσεις κλπ

1. Κλάση Μέσα σε Κλάση

Η *RouteStop* θα μπορούσε να ορισθεί και μέσα στη *Route*:

```
class Route
{
public:
   class RouteStop
   {
   public:
   // . . .
   private:
   // . . .
   }; // RouteStop

   struct RouteStopXptn
   {
   // . . .
   }; // RouteStopXptn

   // . . .
private:
   // . . .
}; // Route

bool operator<( const Route::RouteStop& lhs,
                const Route::RouteStop& rhs );
bool operator>=( const Route::RouteStop& lhs,
                 const Route::RouteStop& rhs );
```

Στην περίπτωση αυτή έχουμε γενικώς την εξής αλλαγή:

όπου είχαμε "`RouteStop`" θα βάζουμε "`Route::RouteStop`" και
όπου είχαμε "`RouteStopXptn`" θα βάζουμε "`Route::RouteStopXptn`"

Αν σου φαίνονται περίεργα ξαναδιάβασε αυτά που λέγαμε για ονοματοχώρους στην §18.5.

Έτσι, για παράδειγμα, ο ορισμός του δημιουργού γίνεται:

```
Route::RouteStop::RouteStop( string aName,
                             float aDist, float aFare )
{
   if ( aName.empty() && (aDist > 0 || aFare > 0) )
      throw Route::RouteStopXptn( "RouteStop",
                                  Route::RouteStopXptn::noName );
   if ( aDist < 0 )
      throw Route::RouteStopXptn( "RouteStop",
                            Route::RouteStopXptn::negDist, aDist );
   if ( aFare < 0 )
      throw Route::RouteStopXptn( "RouteStop",
                            Route::RouteStopXptn::negFare, aFare );
   sName = aName;
   sDist = aDist;
   sFare = aFare;
} // Route::RouteStop::RouteStop
```

Πάντως, στις **throw** μπορούμε να απλουστεύσουμε το γράψιμο, π.χ.:

```
      throw RouteStopXptn( "RouteStop",
                           Route::RouteStopXptn::noName );
```

Στη `main` θα έχουμε:

```
     Route::RouteStop oneStop;
// . . .
        oneRoute.addRouteStop( Route::RouteStop("ΜΑΡΜΑΡΑ",
                                                25.5, 3.7) );
// . . .
        Route::RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
// . . .
        oneRoute.addRouteStop(
               Route::RouteStop( "ΔΙΑΣΤΑΥΡΩΣΗ",
                                 tmp.getDist(), tmp.getFare()) );
// . . .
```

Φυσικά, για να μπορούμε να δώσουμε αυτές τις εντολές θα πρέπει να έχουμε ορίσει τη *RouteStop* σε περιοχή **public**.

Πώς θα επιλέξουμε τη θέση ορισμού μιας τέτοιας κλάσης; Γενικώς: *Αν μια κλάση χρησιμοποιείται από μια μόνον άλλη κλάση αποκλειστικώς είναι προτιμότερο να ορίζεται μέσα στη δεύτερη.*

Στην περίπτωσή μας: η *RouteStop* έπρεπε να ορισθεί μέσα στη *Route* ή έξω από αυτήν; Όπως μπορείς να μαντέψεις, το πρόβλημα αυτού του παραδείγματος και οι κλάσεις που γράψαμε για τη λύση του είναι κομμάτι ενός πιο μεγάλου προβλήματος του οποίου η λύση απαιτεί και άλλες κλάσεις (και αρκετά προγράμματα). Για να απαντήσουμε το ερώτημα θα πρέπει να εφαρμόσουμε τον κανόνα που δώσαμε αλλά σε όλες τις κλάσεις.

2. Μέθοδος *readFromText*

Η *readFromText()* θα έπρεπε να είναι μέθοδος της *RouteStop* και όχι καθολική συνάρτηση. Γιατί τη γράψαμε έτσι; Διότι είναι προχειρογραμμένη σε απαράδεκτο βαθμό. Ως άσκηση (Άσκ. 20-5) σου αφήνουμε τη μετατροπή της *readFromText()* σε μέθοδο και το γράψιμο μιας *readFromText()* για τη *Route*. Για να τη λύσεις θα πρέπει να ξαναδείς τη λύση της Άσκ. 10-8.[10]

3. Χρησιμοποιώντας Εξαιρέσεις

Ένας άλλος τρόπος να γράψουμε το κομμάτι προγράμματος με την εισαγωγή και την αλλαγή ονόματος είναι ο εξής:

```
      try
```

[10] Την έλυσες;

```
      {
         oneRoute.addRouteStop( RouteStop("ΜΑΡΜΑΡΑ", 25.5, 3.7) );

         RouteStop tmp( oneRoute.get1RouteStop("ΣΤΑΥΡΟΣ") );
         oneRoute.deleteRouteStop( "ΣΤΑΥΡΟΣ" );
         oneRoute.addRouteStop( RouteStop("ΔΙΑΣΤΑΥΡΩΣΗ",
                                          tmp.getDist(),
                                          tmp.getFare()) );
      }
      catch( RouteXptn& x )
      {
         switch ( x.errCode )
         {
            case RouteXptn::invalidArgument:
               cout << "Στάση " << x.errStop2.getName()
                    << " υπάρχει" << endl;
               break;
            case RouteXptn::diffName:
               cout << "Υπάρχει στάση στα "
                    << x.errStop2.getDist() << " km με όνομα "
                    << x.errStop2.getName() << endl;
               break;
            case RouteXptn::notFound:
               cout << "Στάση " << x.errStrVal << " δεν υπάρχει"
                    << endl;
               break;
            default:
               throw;
         } // switch
      } // catch
```

Μια **try-catch** μέσα στην **try**! Ναι! Στην περιοχή του **try** έχουμε μόνο τις εντολές που μας ενδιαφέρουν χωρίς τους αμυντικούς ελέγχους με τις **if**. Στην **catch** ασχολούμαστε μόνο με *RouteXptn* και μάλιστα τρία συγκεκριμένα είδη: *invalidArgument*, *diffName* και *notFound*. Για τα υπόλοιπα είδη τι κάνουμε: ξαναρίχνουμε (στη **default**) την εξαίρεση για να την πιάσει η εξωτερική **catch**.

4. Συγκρίσεις Πραγματικών

Να επισημάνουμε τώρα και κάτι που έχει σχέση με προηγούμενα κεφάλαια. Ο τρόπος που επιφορτώσαμε τον τελεστή "`<`" για τη *RouteStop* ορίζει στην πραγματικότητα και την ισότητα για την κλάση αυτή:[11]

$$(a == b) \equiv ((a.sDist - b.sDist) == 0)$$

Δηλαδή δεχόμαστε ως κλειδί το μέλος *sDist*. Θα μπορούσαμε όμως να δεχτούμε και το *sName*. Επιλέξαμε το *sDist* λόγω της ταξινόμησης του πίνακα.

Τώρα πρόσεξε το εξής: Αν προσπαθήσουμε να εισαγάγουμε μια νέα στάση

`RouteStop( "ΣΚΑΡΗ", 25.6, 3.7 )`

είναι φανερό ότι αναφερόμαστε –με άλλο όνομα– στη στάση

`RouteStop( "ΜΑΡΜΑΡΑ", 25.5, 3.7 )`

Η διαφορά των 100 *m* (0.1 *km*) στην απόσταση από την αφετηρία είναι μέσα στα όρια σφάλματος για μια τέτοια μέτρηση. Έτσι, θα ήταν πιο σωστό αν επιφορτώναμε τον "`==`" ως

```
bool operator==( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( fabs(lhs.getDist()-rhs.getDist()) < 0.15 );
} // operator==( const RouteStop
```

(0.15 ή 0.1 ή 0.2, ό,τι διαλέξεις.)

Ο "`<`" –για να είναι συμβατός– θα πρέπει να επιφορτωθεί ως:

```
bool operator<( const RouteStop& lhs, const RouteStop& rhs )
{
   return ( lhs.getDist() - rhs.getDist() < -0.15 );
```

[11] Πώς; Έτσι: (*a* `==` *b*) ≡ (`!`(*a* `<` *b*) `&&` `!`(*b* `<` *a*)).

```
} // operator<( const RouteStop
```

Ακόμη, η σύγκριση:

```
rAllStops[nmNdx].getDist() != aStop.getDist()
```

–στην *addRouteStop()*– θα πρέπει να γραφεί:

```
fabs(rAllStops[nmNdx].getDist()-aStop.getDist()) >= 0.15
```

Παρομοίως, η σύγκριση:

```
rAllStops[nmNdx].getFare() != aStop.getFare()
```

θα πρέπει να γραφεί:

```
fabs(rAllStops[nmNdx].getFare()-aStop.getFare()) >= 0.10
```

ή κάτι παρόμοιο.[12]

5. Μέθοδος *getAllStops()*

Ας πούμε ότι γράφεις τις *Route* και *RouteStop* για κάποιον πελάτη χωρίς να παραδώσεις την αρχική μορφή σε C++. Είναι σίγουρο ότι ο πελάτης σου θα σου ζητήσει τη δυνατότητα να παίρνει τον πίνακα των στάσεων ενός αντικειμένου *Route* για να τον επεξεργαστεί με τα δικά του προγράμματα. Θα χρειαστείς λοιπόν δύο μεθόδους *getAllStops()* και *getNoOfStops()*.

Η δεύτερη είναι απλή:

```
   unsigned int getNoOfStops() const {  return rNoOfStops;  }
```

Η πρώτη χρειάζεται λίγη προσοχή. Το απλούστερο που μπορούμε να κάνουμε είναι να μιμηθούμε τη *c_str()*:

```
const RouteStop* Route::getAllStops() const
{  return rAllStops+1;  }
```

Αυτή η μορφή έχει προβλήματα:

- Θα βάλεις ιδέες στον προγραμματιστή-χρήστη των κλάσεων να αρχίσει να σκαλίζει να ανακαλύψει τα μυστικά μας με τους φρουρούς και –πιθανότατα– να καταστρέψει τα αντικείμενα.
- Αργότερα, που θα αλλάξουμε τη δομή αποθήκευσης σε `set<RouteStop>` τι θα κάνεις;[13]

Μια άλλη ιδέα είναι η εξής: Γράφουμε μια μέθοδο

```
const RouteStop* Route::getAllStops() const
{
   RouteStop* fv( 0 );
   try
   {  fv = new RouteStop[rNoOfStops];  }
   catch( bad_alloc& )
   {  throw RouteXptn( "getAllStops", RouteXptn::allocFailed );  }
   for ( int k(0); k < rNoOfStops; ++k )  fv[k] = rAllStops[k+1];
   return fv;
} // Route::getAllStops
```

που επιστρέφει *αντίγραφο* του πίνακα στάσεων. Αλλά το αντίγραφο είναι δυναμικός πίνακας για τον οποίον δεν φαίνεται η εντολή που παίρνει τη μνήμη και το πιθανότερο είναι ότι ο προγραμματιστής που θα χρησιμοποιήσει τη μεθοδο θα ξεχάσει να βάλει την αντίστοιχη "`delete[]…`".

Μπορούμε να κάνουμε κάτι καλύτερο; Αν σκεφτούμε ότι οι πεπειραμένοι προγραμματιστές μόλις γράψουν μια "`new`" γράφουν και την αντίστοιχη "`delete`" γράφουμε μια μέθοδο

```
void Route::getAllStops( RouteStop*& aStops ) const
{
   try {  renew( aStops, 0, rNoOfStops );  }
   catch( MyTmpltLibXptn& )
   {  throw RouteXptn( "getAllStops", RouteXptn::allocFailed );  }
   for ( int k(0); k < rNoOfStops; ++k )
```

---

[12] Αυτά τα "`0.15`" και "`0.10`" δεν είναι «μαγικές σταθερές»; Είναι και παραείναι! Θα ασχοληθούμε με αυτές αργότερα.

[13] `set<RouteStop>`!!! Τι είναι αυτά; Θα τα μάθεις αργότερα…

```
      aStops[k] = rAllStops[k+1];
} // Route::getAllStops
```

Τώρα, ο πεπειραμένος προγραμματιστής που λέγαμε, θέλοντας να χρησιμοποιήσει τη *getAllStops()* θα γράψει αρχικώς:

```
   RouteStop* allStops( new RouteStop[1] );

   delete[] allStops;
```

και στη συνέχεια θα συμπληρώσει:

```
   RouteStop* allStops( new RouteStop[1] );
// . . .
   oneRoute.getAllStops( allStops );
// . . .
   delete[] allStops;
```

Πάντως θα πρέπει να σημειώσουμε ότι αυτή η «καλύτερη λύση» κοστίζει σε υπολογιστικό χρόνο· αυτά έχουν οι αντιγραφές πινάκων.

6. Γιατί όχι *setCode()*;

Γιατί –ενώ γράψαμε *setFrom()*, *setTo()* και *setInBetween()*– δεν γράψαμε και μια *setCode()* για τη *Route*; Διότι το *rCode* είναι (υποκατάστατο) κλειδί και από τις ΒΔ ξέρουμε ότι δεν αλλάζουμε το κλειδί.[14]

Βέβαια, μπορεί το *rCode* να είναι υποκατάστατο κλειδί αλλά το { *rFrom*, *rTo*, *rInBetween* } είναι το «φυσικό» κλειδί. Γιατί γράψαμε τις *setFrom()*, *setTo()* και *setInBetween()*;

Μπορούμε να μην τις γράψουμε και απλώς να αλλάξουμε τον δημιουργό:

```
Route( int rc=0,
       string aFrom="", string aTo="", string aInBetween="" );
```

Έτσι, στο πρόγραμμά μας θα έχουμε:

```
// . . .
      string   nameF, nameT, nameIB, buf;
      double   dist, fare;

      getline( tin, buf, '\n' );
      getline( tin, nameF, '\t' );
      getline( tin, nameT, '\t' );
      getline( tin, nameIB, '\n' );

      Route oneRoute( atoi(buf.c_str()), nameF, nameT, nameIB );
// . . .
```

Τώρα έγινε χειρότερο! Ας πούμε ότι έβαλα στο *rTo* τιμή `"Δράννα"` αντί για `"Δράμα"`. Δεν μπορώ να το διορθώσω; Στο επόμενο κεφάλαιο θα επιφορτώσουμε τον "`operator=`" για τη *Route*. Θα μπορείς να το διορθώσεις ως εξής:

```
oneRoute = Route( oneRoute.getCode(), oneRoute.getFrom(),
                  "Δράμα", oneRoute.getInBetween() );
```

Φυσικά, κάτι θα πρέπει να κάνεις και για τις στάσεις.

Πάντως, οι *setFrom()*, *setTo()* και *setInBetween()* «νομιμοποιούνται» να υπάρχουν διότι ένας από τους λόγους που εισάγουμε υποκατάστατα κλειδιά είναι ακριβώς αυτός: αν το κλειδί αποτελείται από πολλές (στην περίπτωσή μας: τρεις) λέξεις θα γίνονται λάθη και θα πρέπει να μπορούμε να τα διορθώνουμε.

20.8 Συσχετίσεις Κλάσεων

Η κλάση *Route* –μέρος της λύσης ένός πιο «πραγματικού» προβλήματος– είναι σαφώς πιο πολύπλοκη από τις *Date* και *BString*. Κάθε αντικείμενο κλάσης *Route* περιέχει –δύο ή περισ-

[14] Και για όποιον δεν το κατάλαβε ακόμη: αυτό το παράδειγμα το έχουμε πάρει από μια βάση δεδομένων.

σότερα– αντικείμενα κλάσης *RouteStop*. Έχουμε λοιπόν, στην περίπτωση αυτή, μια **συσχέτιση** (relationship) δύο κλάσεων. Πιο συγκεκριμένα λέμε ότι σε ένα αντικείμενο *Route* έχουμε **σύνθεση** (composition) αντικειμένων *RouteStop*. Κάθε αντικείμενο κλάσης *RouteStop* ανήκει σε ένα αντικείμενο κλάσης *Route*. Όταν καταστρέφεται το αντικείμενο *Route* κατάστρέφονται και τα αντικείμενα κλάσης *RouteStop* που περιέχει.

Η **πολλαπλότητα** (multiplicity) της συσχέτισης είναι **ένα-προς-πολλά** (one-to-many) και τη γράφουμε συμβολικώς 1:*N* ή 1:*. Με μεγαλύτερη ακρίβεια μπορούμε να πούμε:

- Η πολλαπλότητα ως προς τη *RouteStop* ("*N*"): Ένα αντικείμενο κλάσης *Route* έχει κατ' ελάχιστο 2 αντικείμενα κλάσης *RouteStop* (αφετηρία και τέρμα) ενώ δεν υπάρχει περιορισμός στο μέγιστο. Γράφουμε: `2..*`.
- Η πολλαπλότητα ως προς τη *Route* ("1"): Ένα αντικείμενο κλάσης *RouteStop* ανήκει σε ένα ακριβώς αντικείμενο κλάσης *Route* (σε μια διαδρομή και μόνο), με την έννοια που είπαμε πιο πάνω. Γράφουμε: `1..1`.

Για να δουμε και άλλα είδη συσχετίσεων δίνουμε μερικές ακόμη πτυχές του προβλήματος διαδρομών λεωφορείων:

> *Ένα δρομολόγιο γίνεται σε μια διαδρομή, σε συγκεκριμένη ημερομηνία και με συγκεκριμένο χρόνο αναχώρησης (π.χ. 21.06.2015 18:00), από ένα μόνο λεωφορείο και με συγκεκριμένο οδηγό. Σε μερικά δρομολόγια μπορεί να υπάρχει και εισπράκτορας. Τα λεωφορεία έχουν ως ταυτότητα τον αριθμό κυκλοφορίας, ενώ ένα χαρακτηριστικό τους, που μας ενδιαφέρει ιδιαιτέρως, είναι η χωρητικότητά τους (αριθμός θέσεων).*

Μεγάλο μέρος της λύσης του προβλήματος διαχείρισης των δρομολογίων είναι η σχεδίαση και η υλοποίηση μιας Βάσης Δεδομένων (ΒΔ), όπου θα βρίσκονται όλα τα δεδομένα που μας ενδιαφέρουν. Εδώ θα ασχοληθούμε με τα προγράμματα για τη διαχείριση των στοιχείων που βρίσκονται στη ΒΔ και πιο συγκεκριμένα με τις κλάσεις που θα έχουμε σε αυτά τα προγράμματα. Στα προγράμματα αυτά υπάρχουν *συλλογές* με αντικείμενα των κλάσεων οι οποίες επικοινωνούν με τη ΒΔ. Όταν εκτελείται κάποιο από αυτά τα προγράμματα φορτώνει από τη ΒΔ στη μνήμη –στις συλλογές– τα δεδομένα που χρειάζεται. Στο τέλος της εκτέλεσης κάποιου προγράμματος, οι οποιεσδήποτε ενημερώσεις των περιεχομένων των συλλογών έχουν μεταφέρθεί και στη ΒΔ.

Παρατηρήσεις: ▶

1. Κατά τη διάρκεια της εκτέλεσης έχουμε φορτωμένο στη μνήμη ένα τμήμα της ΒΔ. Πόσο μεγάλο τμήμα; Από μερικά αντικείμενα μέχρι ολόκληρη τη ΒΔ.
2. Παραπάνω λέμε ότι τα αντικείμενα υπάρχουν σε *συλλογές*. Τι είδους συλλογές; πίνακες ή δένδρα ή λίστες κλπ.
3. Η απάντηση στα παραπάνω ερωτήματα καθορίζεται από την πολιτική διαχείρισης της ΒΔ από το πρόγραμμα. Η επιλογή πολιτικής δεν γίνεται στην τύχη. Χρειάζεται «μέτρημα»!

◀

Από την περιγραφή που μας δίνεται μπορούμε να δούμε τις εξής κλάσεις:

- Για τις στάσεις:

```
class RouteStop
```

- Για τις διαδρομές:

```
class Route
```

- Για τους εργαζόμενους:

```
class Employee
{
public:
// . . .
private:
   unsigned int eIdNum;      // αριθμός μητρώου εργαζομένου
   string       eSurname;    // επώνυμο
```

```
   string       eFirstName;  // όνομα
   char         eOccupation; // εργασία
}; // Employee
```

- Για τα λεωφορεία:

```
class Bus
{
public:
// . . .
private:
   string        bRegNum;      // αριθμός κυκλοφορίας
   unsigned int  bNoOfSeats;   // αριθμός θέσεων
   unsigned int* bAllServices; // πίνακας δρομολογίων
}; // Bus
```

- Για τα δρομολόγια:

```
class BusService
{
public:
// . . .
private:
   unsigned int bseCode;      // κωδικός δρομολογίου
   Route        bseRoute;     // διαδρομή
   DateTime     bseDT;        // ημερομηνία/ώρα αναχώρησης
   Bus          bseBus;       // λεωφορείο
   Employee     bseDriver;    // οδηγός
   Employee     bseConductor; // εισπράκτορας
}; // BusService
```

Αυτά για μια πρώτη καταγραφή. Τώρα ας τα σκεφτούμε λίγο παραπάνω.

- Ας πάρουμε ως διαδρομή το παράδειγμα που δίνεται στην περιγραφή του προβλήματος: Αθήνα – Σούνιο από τα Μεσόγεια. Πόσα τέτοια δρομολόγια γίνονται κάθε μέρα; Πολλά! Μπορεί και ένα κάθε δυο ώρες. Προφανώς, το να έχουμε σε κάθε ένα από αυτά τα δρομολόγια όλα τα στοιχεία και κυρίως όλες τις στάσεις της διαδρομής είναι σπατάλη μνήμης. Εκτός από αυτό όμως, εδώ έχουμε μια περίπτωση **πλεονασμού** (redundancy) που δημιουργεί τις λεγόμενες **ανωμαλίες ενημέρωσης** (update anomalies). Θα πρέπει να διορθώνεις *όλα τα δρομολόγια* που έχουν τη συγκεκριμένη διαδρομή κάθε φορά που θέλεις:
  - Να *τροποποιήσεις* κάποιο στοιχείο της διαδρομής· αυτή είναι η **ανωμαλία τροποποίησης** (modification anomaly).
  - Να *διαγράψεις* μια στάση· αυτή είναι η **ανωμαλία διαγραφής** (deletion anomaly).
  - Να *εισαγάγεις* μια (νέα) στάση· αυτή είναι η **ανωμαλία εισαγωγής** (insertion anomaly).
- Παρόμοια προβλήματα υπάρχουν με τα στοιχεία του λεωφορείου, του οδηγού και του εισπράκτορα.

Πώς αντιμετωπίζονται αυτά τα προβλήματα; Με το να γράψουμε την κλάση για τα δρομολόγια ως εξής:

```
class BusService
{
public:
// . . .
private:
   unsigned int bseRouteCode;      // κωδικός διαδρομής
   DateTime     bseDT;             // ημερομηνία/ώρα αναχώρησης
   unsigned int bseBusRegNum;      // αρ. κυκλοφορίας λεωφορείου
   unsigned int bseDriverIdNum;    // αρ. μητρ. οδηγού
   unsigned int bseConductorIdNum; // αρ. μητρ. εισπράκτορα
}; // BusService
```

Δηλαδή αντί να βάλουμε μέσα σε ένα αντικείμενο κλάσης *BusService* ολόκληρα τα αντικείμενα των άλλων κλάσεων βάλαμε μόνον τα κλειδιά των αντικειμένων.[15]

Γιατί δεν κάναμε το ίδιο και για τις στάσεις; Εκεί τα πράγματα είναι διαφορετικά: Φυσικά μια στάση μπορεί να ανήκει σε πολλές διαδρομές αλλά δεν είναι απαραίτητο να έχει σε όλες τις διαδρομές την ίδια απόσταση από την αφετηρία –που μπορεί να μην είναι ίδια για όλες τις διαδρομές– και επομένως τό ίδιο κόμιστρο.

Και τώρα να δούμε και λίγη ορολογία για όσα είδαμε:

- Η σχέση της *BusService* με τις *Route*, *Bus* και *Employee* είναι σχέση **συνάθροισης** (aggregation). Κάθε αντικείμενο κλάσης *Route*, *Bus* και *Employee* μπορεί να ανήκει σε *ένα ή περισσότερα* αντικείμενα κλάσης *BusService*. Όταν καταστρέφεται το αντικείμενο *BusService δεν* καταστρέφονται τα αντικείμενα *Route*, *Bus* και *Employee* που συναθροίζει.
  - *BusService–Route*. Πολλαπλότητα ως προς τη *BusService*: Σε μια διαδρομή εκτελούνται από κανένα (ανενεργή διαδρομή) μέχρι πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς τη *Route*: Ένα δρομολόγιο εκτελείται σε μια ακριβώς διαδρομή: `1..1`. (Συσχέτιση **πολλά-προς-ένα**, many-to-one.)
  - *BusService–Bus*. Πολλαπλότητα ως προς τη *BusService*: Ένα λεωφορείο εκτελεί από κανένα μέχρι πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς τη *Bus*: Ένα δρομολόγιο εκτελείται «*από ένα μόνο λεωφορείο*»: `1..1`. (Συσχέτιση πολλά-προς-ένα.)
  - *BusService–Employee*. Οι κλάσεις αυτές συσχετίζονται με δύο τροπους που ξεχωρίζουν από τον **ρόλο** (role) του εργαζόμενου. Πολλαπλότητα –και στις δύο περιπτώσεις– ως προς τη *BusService*: Ένας εργαζόμενος εργάζεται σε κανένα, σε ένα ή σε πολλά δρομολόγια: `0..*`. Πολλαπλότητα ως προς την *Employee*: α) αν ο ρόλος του εργαζόμενου είναι οδηγός (συσχέτιση με την τιμή του μέλους *bseDriverIdNum*): ένα δρομολόγιο εκτελείται από έναν οδηγό: `1..1` β) αν ο ρόλος είναι εισπράκτορας (συσχέτιση με την τιμή του μέλους *bseConductorIdNum*, μπορεί να μην υπάρχει): `0..1`. (Συσχετίσεις πολλά-προς-ένα.)

Η *σύνθεση* και η *συνάθροιση* είναι δύο περιπτώσεις της σχέσης "`has_a`". Στη βιβλιογραφία μπορεί να συναντήσεις και την αντίστροφη σχέση της "`has_a`", την "`is_part_of`". Γράφουμε δηλαδή:

Route `has_a` *RouteStop* ή *RouteStop* `is_part_of` *Route*
BusService `has_a` *Bus* ή *Bus* `is_part_of` *BusService*

Οι κλάσεις που είδαμε παραπάνω είναι όλες κλάσεις στη **φάση της υλοποίησης** (implementation phase). Η σχεδίαση των κλάσεων όμως ξεκινάει πολύ νωρίς στη φάση της **σχεδίασης** (design) και περνάει από –συνήθως αρκετές– αναθεωρήσεις μέχρι να φτάσει σε τελική μορφή.

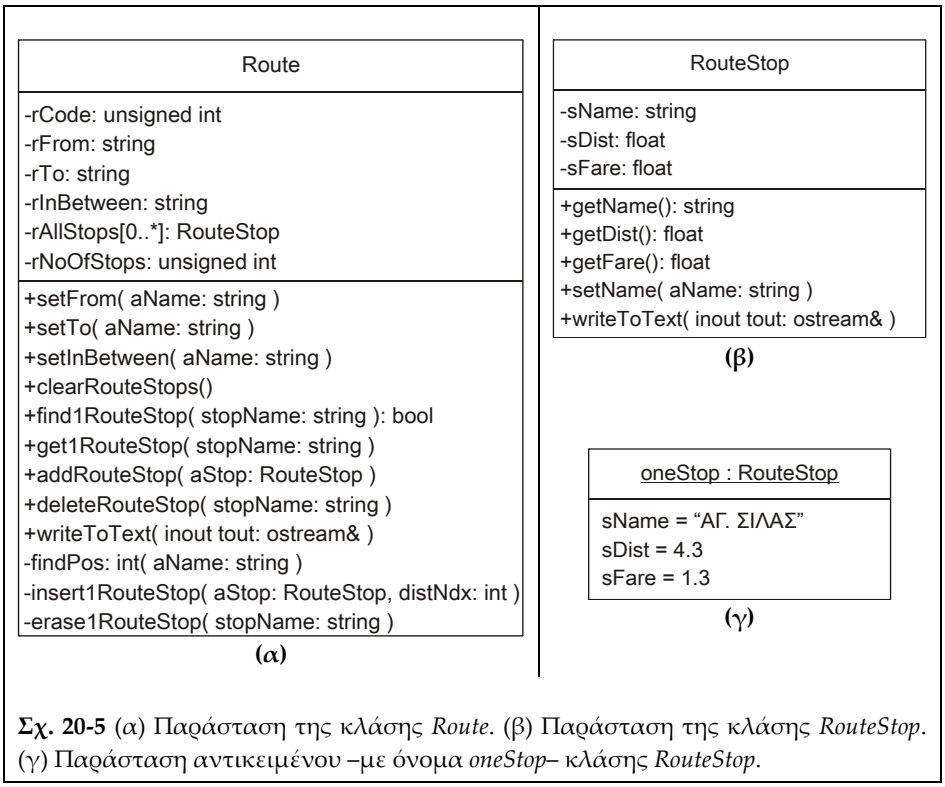
Οι συσχετίσεις που είδαμε παραπάνω είναι διμελείς και απλές. Στη φάση της σχεδίασης οι συσχετίσεις μπορεί να είναι πολυμελείς και πιο πολύπλοκες. Μια τέτοια συσχέτιση μπορεί να συνοδεύεται και από αντίστοιχη **κλάση συσχέτισης** (association class).

Η κλάση *BusService* είναι στην πραγματικότητα η κλάση μιας *τετραμελούς συσχέτισης* των κλάσεων *Route*, *Bus*, *Employee* και *Employee*. Έτσι, έχει ως μέλη τα χαρακτηριστικά (κλειδιά) των κλάσεων που συσχετίζει: *bseRouteCode*, *bseBusRegNum*, *bseDriverIdNum* και *bseConductorIdNum*. Έχει όμως και ένα επιπλέον χαρακτηριστικό το *bseDT*.

Για να δούμε άλλη μια κλάση συσχέτισης ξεκινούμε με άλλη μια πτυχή του προβλήματός μας:

> *Για κάθε στάση κάποιας από τις διαδρομές που εξυπηρετούν τα λεωφορεία του ΚΤΕΛ καταγράφουμε: το όνομα (που είναι μοναδικό), περιγραφή που καθορίζει*

[15] Λέμε ότι αυτά τα χαρακτηριστικά της *BusService* είναι **ξένα κλειδιά** (foreign keys).



Σχ. 20-5 (α) Παράσταση της κλάσης *Route*. (β) Παράσταση της κλάσης *RouteStop*. (γ) Παράσταση αντικειμένου –με όνομα *oneStop*– κλάσης *RouteStop*.

με ακρίβεια τη θέση και το επίπεδο υποδομών για την εξυπηρέτηση των επιβατών (σταθμός, οικίσκος, στέγαστρο, σήμα στάσης).

Κάθε στάση παριστάνεται με ένα αντικείμενο κλάσης

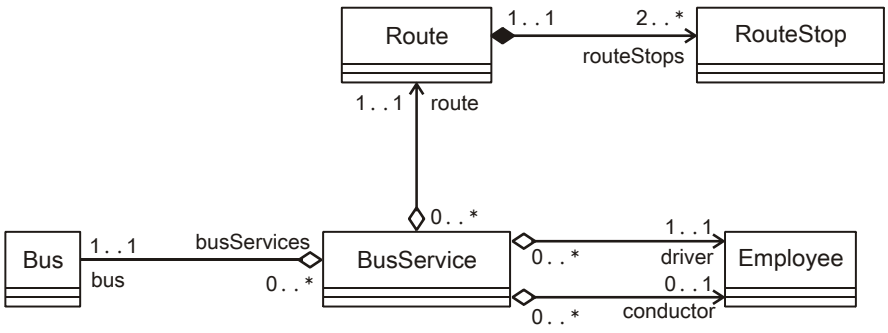
```
class BusStop
{
public:
// . . .
private:
   string        bsName;      // όνομα στάσης
   string        bsLocation;  // θέση
   unsigned char bsFaciLevel; // επίπεδο εξυπηρέτησης
}; // BusStop
```

Η *BusStop* συσχετίζεται με τη *Route*. Κάθε διαδρομή (αντικείμενο κλάσης *Route*) έχει 2 ή περισσότερες στάσεις (αντικείμενα κλάσης *BusStop*)· πολλαπλότητα 2..*. Κάθε στάση περιλαμβάνεται σε μια τουλάχιστον διαδρομή: πολλαπλότητα 1..*. Η πολλαπλότητα της συσχέτισης είναι πολλα-προς-πολλά (*M*:*N*). Για κάθε στάση που συσχετίζουμε με μια διαδρομή υπολογίζουμε την απόστασή της και το κόμιστρο από την αφετηρία. Δηλαδή η συσχέτιση έχει δύο χαρακτηριστικά. Έχουμε λοιπόν μια κλάση συσχέτισης, τη *RouteStop*, που έχει ως μέλη τα κλειδιά των κλάσεων που συσχετίζονται (*rCode*, *bsName*) και επιπλέον τα *sDist* και *sFare*. Στη φάση της υλοποίησης, κάνοντας την επιλογή να βάλουμε μέσα στο αντικείμενο *Route* όλα τα αντικείμενα *RouteStop* που το αφορούν, ο κωδικός διαδρομής είναι άχρηστος.

Αν έχεις ασχοληθεί με τη σχεδίαση Βάσεων Δεδομένων τα παραπάνω θα σου είναι οικεία. Εκεί, στα μοντέλα Οντοτήτων-Συσχετίσεων, έχεις *οντότητες* (entities)· οι κλάσεις είναι ένας τρόπος υλοποίησης των οντοτήτων.

20.8.1 Διαγραμματικές Παραστάσεις

Στην Τεχνολογία Λογισμικού χρησιμοποιούνται πολλά είδη διαγραμματικών παραστασεων που βοηθούν τον μηχανικό λογισμικού στις διάφορες φάσεις της ανάλυσης, της σχεδία-



Σχ. 20-6 Διαγραμματική παράσταση των κλάσεων και των συσχετίσεών τους.

σης, της ανάπτυξης κλπ. Εδώ θα πάρουμε μια ιδέα ενός είδους διαγραμμάτων. Δεν είναι κατ' ανάγκην το πιο χρήσιμο· το επιλέγουμε επειδή παριστάνει με διαγράμματα αυτά που είπαμε παραπάνω. Ακολουθούμε (όχι με «ευλάβεια») τούς κανόνες της UML (OMG 2010).[16]

Να ξεκινήσουμε με την παράσταση κλάσεων. Αυτή γίνεται με ένα ορθογώνιο με τρία τμήματα (Σχ. 20-5): Στο πρώτο γράφουμε το όνομα της κλάσης, στο δεύτερο τα μέλη και στο τρίτο τις μεθόδους. Σε ορισμένες περιπτώσεις μπορεί να υπάρχει και τέταρτο τμήμα με τις εξαιρέσεις που ρίχνουν οι μέθοδοι της κλάσης.

Κάθε μέλος γράφεται σε ξεχωριστή σειρά με το εξής συντακτικό:

ορατότητα, *όνομα*, "**:**", *τύπος*

όπου:

ορατότητα = "**+**" | "**-**" | "**#**";

Το "**+**" σημαίνει ότι δηλώνεται σε περιοχή "**public**", το "**-**" σε περιοχή "**private**" και το "**#**" σε περιοχή "**protected**" (αυτή θα τη μάθουμε αργότερα).

Για τις μεθόδους γράφουμε:

ορατότητα, *όνομα*, "**(**", [*λίστα παραμέτρων*]. "**)**", "**:**", *τύπος*

όπου:

λίστα παραμέτρων = *παράμετρος* | *λίστα παραμέτρων*, *παράμετρος*;

παράμετρος = [*είδος*], *όνομα*, [*πολλαπλότητα*]. "**:**", *τύπος*;

είδος = "**in**" | "**out**" | "**inout**";

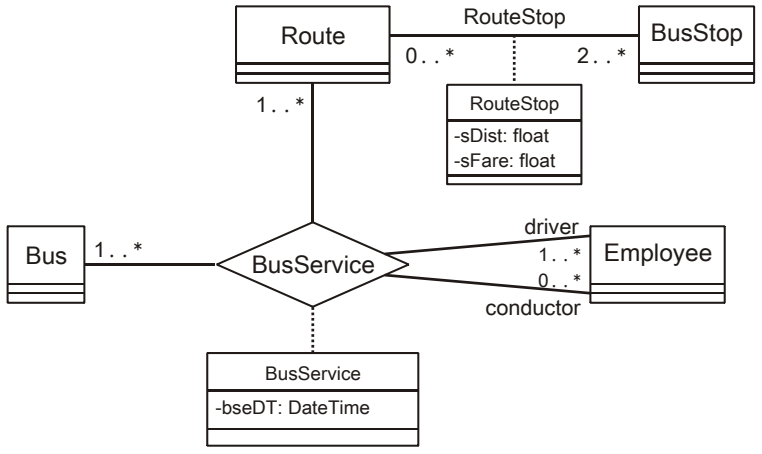
Αν δεν υπάρχει το *είδος* εννοείται "**in**".

Στα διαγράμματα δεν είναι απαραίτητο να βάζουμε όλα τα στοιχεία μιας κλάσης· βάζουμε αυτά που απαιτούνται για να περιγράψουμε ό,τι θέλουμε. Μπορείς να παραλείψεις και ολόκληρο τμήμα. Στο Σχ. 20-6 βλέπεις ότι από τις κλάσεις που παριστάνονται φαίνεται μόνον το τμήμα του ονόματος ενώ στο Σχ. 20-7 βλέπεις τις κλάσεις συσχέτισης με τα τμήματα ονόματος και μελών.

Να σχολιάσουμε τώρα αυτά που βλέπουμε στο Σχ. 20-6:

- Όπως είπαμε, στη συσχέτιση *Route-RouteStop* έχουμε *σύνθεση* αντικειμένων κλάσης *RouteStop* σε ένα αντικείμενο κλάσης *Route*. Αυτό παριστάνεται με τον μαύρο ρόμβο στο άκρο προς τη *Route*. Στο ίδιο άκρο βλέπουμε την πληθικότητα της κλάσης στη συσχέτιση και τον ρόλο της. Στο άλλο άκρο βλέπουμε ένα βέλος προς τη *RouteStop*. Αυτό δείχνει μια **μονόδρομη** (uni-directional) συσχέτιση: Στην τιμή ενός αντικειμένου *Route* υπάρχει η πληροφορία για τα αντικείμενα *RouteStop* με τα οποία συσχετίζεται (πίνακας rAllStops) ενώ στην τιμή ενός αντικειμένου *RouteStop* δεν υπάρχει πληροφορία για σχετιζόμενο αντικείμενο *Route*. Για τον λόγο αυτόν δεν υπάρχει και ρόλος για το αντικείμενο *RouteStop*.

[16] UML: Unified Modeling Language, OMG: Object Management Group.



Σχ. 20-7 Το διάγραμμα οντοτήτων-συσχετίσεων από το οποίο προήλθε το διάγραμμα του Σχ. 20-6. Εδώ η *RouteStop* είναι κλάση (διμελούς) συσχέτισης. Κλάση (τετραμελούς) συσχέτισης είναι και η *BusService*. Οι πληθικότητες διαφέρουν από αυτές του Σχ. 20-6.

- Στη συσχέτιση *BusService–Bus* έχουμε –στο άκρο προς τη *BusService*– έναν λευκό ρόμβο. Αυτός δείχνει ότι ένα αντικείμενο κλάσης *Bus συναθροίζεται* σε ένα αντικείμενο κλάσης *BusService*. Στη γραμμή συσχέτισης δεν υπάρχουν βέλη· αυτό δείχνει μια **αμφίδρομη** (bi-directional) συσχέτιση. Πράγματι, από ένα αντικείμενο κλάσης *BusService* μπορούμε να βρούμε το λεωφορείο που την εξυπηρετεί και από ένα αντικείμενο *Bus* μπορούμε να βρούμε τα δρομολόγια που εξυπηρετεί (πίνακας bAllServices). Αυτό φαίνεται και από τους ρόλους που βλέπουμε και στα δύο άκρα.
- Στις παραπάνω συσχετίσεις οι αναγραφές των ρόλων μπορεί να μην είναι απαραίτητες. Αυτό όμως δεν ισχύει για τις δύο συσχετίσεις *BusService–Employee*. Χωρίς τις αναγραφές των ρόλων δεν είναι φανερές οι αντιστοιχίες των (διαφορετικών) πληθικοτήτων.

Στο Σχ. 20-7 βλέπεις το διάγραμμα κλάσεων από το οποίο προήλθε αυτό του Σχ. 20-6. Πρόκειται για ένα **διάγραμμα οντοτήτων-συσχετίσεων** (entity-relationship –ER– diagram). Εδώ βλέπεις:

- Μια επιπλέον κλάση, τη *BusStop*, που δεν υπάρχει στο Σχ. 20-6.
- Η *RouteStop* είναι μια κλάση συσχέτισης που «κρέμεται» από τη γραμμή της (διμελούς) συσχέτισης με το ίδιο όνομα. Παρομοίως, η *BusService* είναι η κλάση της τετραμελούς συσχέτισης με το ίδιο όνομα. Ο ρόμβος χρησιμοποιείται για την παράσταση συσχετίσεων με τρία ή περισσότερα μέλη.[17]
- Οι πληθικότητες έχουν αλλάξει. Πώς υπολογίζονται αυτές της *RouteStop*; Πληθικότητα της *Route* είναι η απάντηση στην ερώτηση: «*Πόσες στάσεις έχει μια διαδρομή;*» ενώ πληθικότητα της *BusStop* είναι η απάντηση στην ερώτηση: «*Σε πόσες διαδρομές συμμετέχει μια στάση;*» Για τις πληθικότητες της *BusService* οι ερωτήσεις είναι πιο πολύπλοκες. Για παράδειγμα για την πληθικότητα της *Route* ρωτάμε: «*Ένα λεωφορείο, με έναν όδηγό και (πιθανόν) με έναν εισπράκτορα πόσες διαδρομές εξυπηρετεί;*» Με παρόμοιες ερωτήσεις βρίσκουμε τις πληθικότητες των άλλων κλάσεων *Bus*, *Employee* (*driver*) και *Employee* (*conductor*)

Το **διάγραμμα κλάσεων** (class diagram) είναι ένα είδος στατικού **διαγραμμάτος δομής** (structure diagram) από τα πολλά που υπάρχουν. Πέρα από αυτά υπάρχουν και τα

[17] Για τις συσχετίσεις θα δεις να χρησιμοποιούνται συνήθως *ρήματα* και όχι ουσιαστικά.

διαγράμματα συμπεριφοράς (behavior diagrams) που περιγράφουν τα μηνύματα που μπορεί να δεχθεί κάποιο αντικείμενο και την ανταπόκρισή του σε αυτά.

Για να καταλάβεις τη χρησιμότητα αυτών των διαγραμμάτων θα πρέπει να πάρεις υπόψη σου ότι πρώτα θα γίνει το διάγραμμα οντοτήτων-συσχετίσεων, από αυτό θα προέλθει –με χρήση κάποιων κανόνων– το διάγραμμα κλάσεων του Σχ. 20-6 και από αυτό (και από άλλα) θα σχεδιαστούν και θα αναπτυχθούν οι κλάσεις. Έτσι, τα διαγράμματα μπορεί να χρησιμοποιηθούν για τον έλεγχο ορθότητας της σχεδίασης: κάποιοι, που είναι ειδικοί στο πρόβλημα που λύνει το πληροφοριακό σύστημα που σχεδιάζεις, θα καταλάβουν πολύ πιο εύκολα ένα τέτοιο διάγραμμα παρά το πρόγραμμα που γράφεις και έτσι θα μπορείς να έχεις τις διορθώσεις τους και τις παρατηρήσεις τους. Στη συνέχεια, είναι δική σου ευθύνη να απεικονίσεις με σωστό τρόπο αυτά που δείχνει το διάγραμμα στο πρόγραμμα που γράφεις.

20.9 Για να Γράψουμε μια Κλάση...

Όταν σχεδιάζουμε μια κλάση το εύκολο μέρος είναι να βρούμε τα χαρακτηριστικά των αντικειμένων της. Το πολύ-πολύ να κάνουμε μερικές υπερβολές και να βάλουμε ως χαρακτηριστικά κάποιες υπολογιζόμενες ποσότητες. Για παράδειγμα στη *Date* αν βάζαμε ως μέλος:

```
WeekDay dDayOfWeek;
```

θα ήταν λάθος αφού αυτή μπορεί να υπολογιστεί.

Πάντως τέτοια λάθη εύκολα εντοπίζονται και διορθώνονται.

Η σχεδίαση συμπεριφοράς των αντικειμένων, δηλαδή το πώς θα ανταποκρίνονται στα διάφορα μηνύματα, θέλει περισσότερη προσοχή. Στις περισσότερες περιπτώσεις, όπως είναι φυσικό, τα πιο σημαντικά στοιχεία της συμπεριφοράς έχουν να κάνουν με τη συγκεκριμένη κλάση και τις εφαρμογές που αυτή χρησιμοποιείται. Υπάρχουν όμως και μερικά στοιχεία τα οποία, αν δεν είναι υποχρεωτικό να υπάρχουν, θα πρέπει να εξετασθεί η ανάγκη ύπαρξής τους.

Αυτά θα τα τυποποιήσουμε αργότερα σε μια «συνταγή». Προς το παρόν όμως ας σκεφθούμε μερικά από αυτά.

Ο **ερήμην δημιουργός**, με ή χωρίς αρχικές τιμές, είναι απαραίτητος για τη δήλωση μεταβλητών-αντικειμένων της κλάσης (και όχι μόνον). Τον γράφουμε έτσι ώστε αρχικώς το αντικείμενο να συμμορφώνεται με την αναλλοίωτη της κλάσης.

Αν τα αντικείμενα της κλάσης χρειάζονται δυναμική μνήμη τότε πρέπει να ορίσουμε εμείς **δημιουργό αντιγραφής**, **τελεστή εκχώρησης** και **καταστροφέα**.[18]

Τέλος, χρειαζόμαστε μεθόδους για να παίρνουμε (*get*) και να αλλάζουμε (*set*) τις τιμές των μελών ενός αντικειμένου. Όλων; Όχι κατ' ανάγκη. Για παράδειγμα, στη *BString*, έχει νόημα να γράψουμε μια *getReserved()*; Όχι βέβαια! Η ύπαρξη του μέλους *bsReserved*

- *έχει σχέση* με τον τρόπο που αποφασίσαμε εμείς να χειριστούμε την παραχώρηση δυναμικής μνήμης στα αντικείμενά μας και
- *δεν έχει σχέση* με το «φυσικό νόημα» των αντικειμένων κλάσης *BString*.

Πολύ περισσότερο, δεν έχει νόημα να γράψουμε μια *setReserved*(. . .).

Σε πολλές περιπτώσεις οι απαιτήσεις της εφαρμογής μας οδηγούν να γράψουμε μεθόδους που είναι πολύ προτιμότερες από τις "*get*" και "*set*". Αν γυρίσουμε στη *Battery*, του προηγούμενου κεφαλαίου, οι *powerDevice()* και *maxTime()* είναι πολύ πιο ουσιαστικές από μια *getEnergy()* και η *recharge()* πιο ουσιαστική από μια *setEnergy()*. Η *Τεχνολογία Λογισμικού* –και πιο συγκεκριμένα η *Αντικειμενοστρεφής Σχεδίαση*– μας δίνει μια τεχνική, τις **περι-**

[18] Αυτά είναι τα μέλη του «κανόνα των τριών» που θα δούμε στη συνέχεια.

πτώσεις χρήσης (use cases), που –μεταξύ άλλων– μας βοηθάει να βρούμε τέτοιες «καλές» μεθόδους.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

20-1 Στέλνουμε μήνυμα σε ένα αντικείμενο:

- Γράφοντας το όνομα της κλάσης και το όνομα της αντίστοιχης μεθόδου με τα κατάλληλα ορίσματα
- Γράφοντας το όνομα του αντικειμένου και το όνομα της αντίστοιχης μεθόδου με τα κατάλληλα ορίσματα
- Με eMail

20-2 Είναι απαραίτητο να ορίσουμε τελεστή εκχώρησης για μια κλάση αν

- Αν κάθε αντικείμενό της έχει έναν τουλάχιστον δυναμικό πίνακα
- Αν κάθε αντικείμενό της έχει έναν πίνακα τουλάχιστον
- Αν κάθε αντικείμενό της έχει περισσότερα από ένα μέλη

20-3 Χρειάζεται να ορίσουμε εμείς τελεστή εκχώρησης για την κλάση *RouteStop*;

- Ναι (δικαιολόγησε την απάντησή σου και όρισέ τον)
- Όχι (δικαιολόγησε την απάντησή σου)

Το ίδιο για την κλάση:

```
class RouteStopCStr
{
public:
// . . .
private:
   char  sName[20]; // όνομα στάσης
   float sDist;     // απόσταση από αφετηρία σε km
   float sFare;     // τιμή εισιτηρίου από την αφετηρία
}; // RouteStopCStr
```

Β Ομάδα

20-4 Συμπληρώνοντας αυτά που λέμε στην πρώτη παρατήρηση της §20.7.5, ξαναγράψε ολόκληρο το πρόγραμμα των διαδρομών λεωφορείου ορίζοντας τη *RouteStop* μέσα στη *Route*.

20-5 Μετάτρεψε τη *readFromText()* σε μέθοδο της *RouteStop*. Όπως έχουμε τονίσει, ένα αρχείο text μπορεί να έχει πολλά λάθη (αφού ο χρήστης μπορεί να προσπαθήσει να το αλλάξει με έναν απλό κειμενογράφο). Βάλε άμυνες για όσο περισσότερες «κακοτοπιές» μπορείς. Πριν ξεκινήσεις, ξαναδές τη λύση της Άσκ. 10-8.

Το ίδιο προσεκτικά γράψε μια *readFromText* για τη *Route*.

Γ Ομάδα

20-6 Μια βιομηχανία κατασκευάζει έπιπλα. Για λόγους προγραμματισμού της παραγωγής και προμήθειας πρώτων υλών η βιομηχανία κρατάει τη σύνθεση του κάθε προϊόντος της σε ένα αντικείμενο κλάσης:

```
class Product
```

```
{
public:
// . . .
private:
   string       prCode;      // Κωδικός προϊόντος
   string       prDescr;     // Περιγραφή
   Component*   prComp;      // Συνιστώσες
   unsigned int prNoOfComp;  // Πλήθος συνιστωσών
}; // Person
```

Ο πίνακας των συνιστωσών (πρώτων υλών) έχει στοιχεία τύπου:

```
class Component
{
public:
// . . .
private:
   char   cCode[10];  // Κωδικός πρώτης ύλης
   double cQty;       // ποσότητα
}; // Product
```

1. Υλοποίησε την κλάση με τις μεθόδους που θεωρείς απαραίτητες.

2. Ας πούμε ότι θέλουμε να αλλάξουμε την ποσότητα κάποιας πρώτης ύλης (ξέρουμε τον κωδικό της) που υπάρχει στη σύνθεση κάποιου προϊόντος. Γράψε μέθοδο που θα κάνει αυτήν τη δουλειά.

3. Πώς θα μπορούσαμε να γράψουμε ένα αντικείμενο κλάσης *Product* σε ένα μη μορφοποιημένο αρχείο ώστε να έχουμε τη δυνατότητα να το ξαναδιαβάσουμε στη συνέχεια. Γράψε μεθόδους *save()* και *load()* που λύνουν το πρόβλημα αυτό.

Αν θέλεις, μπορείς να εισαγάγεις και άλλα μέλη στην κλάση· δεν μπορείς όμως να αφαιρέσεις ή να αλλάξεις αυτά που υπάρχουν.