Μέρος Γ
Αντικειμενοστρεφής Προγραμματισμός

19. Από τις Δομές στις Κλάσεις 601
project 3: Φοιτητές και Μαθήματα 633
20. Κλάσεις και Αντικείμενα - Βασικές Έννοιες 655
21. Ειδικές Συναρτήσεις και Άλλα 696
22. Επιφόρτωση Τελεστών 739
project 4: Φοιτητές και Μαθήματα Αλλιώς 771
project 5: Σύνολα Γραμμάτων 825
23. Κληρονομιές 841
project 6: Φοιτητές και Μαθήματα με Κληρονομιές 885
24. «Πέφτεις σε Λάθη…» - Εξαιρέσεις 917
25. Περιγράμματα Κλάσεων 951
26. Βιβλιοθήκη Παγίων Περιγραμμάτων - Standard Template Library (STL) 989
project 7: Φοιτητές και Μαθήματα με STL 1039
project 8: Προβλήματα για Λύση 1067

κεφάλαιο

19

Από τις Δομές στις Κλάσεις

Habeas Corpus

Ο στόχος μας σε αυτό το κεφάλαιο:

- Θα θυμηθούμε τις δομές.
- Θα δούμε προβλήματα που υπάρχουν και πώς λύνονται με τις κλάσεις.

Προσδοκώμενα αποτελέσματα:

Όταν θα έχεις μελετήσει αυτό το κεφάλαιο θα μπορείς να σχεδιάσεις και να υλοποιήσεις μια απλή κλάση.

Έννοιες κλειδιά:

- *δομή*
- *κλάση*
- *εσωτερικά* (`private`) *μέλη*
- *ανοικτά* (`public`) *μέλη*
- *είδος των μεθόδων*

Περιεχόμενα:

19.1 Κλάσεις ... 602
- 19.1.1 "`const`" ... 609
- 19.1.2 Βοηθητικές Συναρτήσεις ... 609
- 19.1.3 "`class`" και "`public`" ... 610
- 19.1.4 Επιφόρτωση Τελεστών ... 612
- 19.1.5 Ονοματολογία ... 613

19.2 Το Είδος των Μεθόδων ... 613

19.3 Κατανομή σε Αρχεία ... 613
- 19.3.1 Τα «Μυστικά» της Υλοποίησης ... 614

19.4 Μέθοδοι "`inline`" ... 615

19.5 Αναλλοίωτη της Κλάσης - Κλάσεις Εξαιρέσεων ... 615

19.6 "`class`" ή "`struct`"; ... 617

19.7 Από τη "`struct GrElmn`" στην "`class GrElmn`" ... 619

19.8 Μια Κλάση για Μπαταρίες ... 625
- 19.8.1 Μέθοδοι "*get*", "*set*" ... 626
- 19.8.2 Μέθοδος *powerDevice()* ... 626
- 19.8.3 Μέθοδος *maxTime()* ... 627
- 19.8.4 Μέθοδος *reCharge* ... 628
- 19.8.5 Η Κλάση μας Τελικώς ... 628
- 19.8.6 Το Πρόγραμμα ... 629

19.9 Τι (Πρέπει Να) Έμαθες στο Κεφάλαιο Αυτό ... 630

Ερωτήσεις - Ασκήσεις ... 631
- Α Ομάδα ... 631
- Β Ομάδα ... 631
- Γ Ομάδα ... 632

Εισαγωγικές Παρατηρήσεις:
Ας ξεκινήσουμε από το "*Habeas Corpus*". Κυριολεκτικώς σημαίνει «να έχεις το σώμα σου» (να είσαι κύριος του σώματός σου). Αυτή είναι μια αρχή δικαίου που σήμερα σημαίνει ότι κάθε πολίτης έχει δικαίωμα να δικάζεται από τον «φυσικό δικαστή του». Η διατύπωσή της όμως έχει σχέση με τα κρατούντα την εποχή της φεουδαρχίας· ο φεουδάρχης ήταν απόλυτος κύριος του κάθε υπηκόου του (και του σώματός του) αφού, σε οποιαδήποτε διένεξη, δικαστής ήταν ο φεουδάρχης! Το "*Habeas Corpus*" κατακτήθηκε με την επανάσταση του Cromwell το 1679.

Και τι σχέση έχει με τον προγραμματισμό; Οι μεταβλητές (αντικείμενα) που ο τύπος τους είναι απλή δομή είναι στο «έλεος» του προγράμματος που τις χρησιμοποιεί (**εφαρμογή-πελάτης**, client application). Έτσι:

- Όπως λέγαμε στην §15.3 «*Ο δημιουργός δεν μας απαγορεύει να δηλώσουμε:*

```
Date d1( 2004, 14, 37 );
```

  Θα πρέπει να τον εφοδιάσουμε με μερικούς ελέγχους ώστε να απαγορεύει τέτοια λάθη.»
- Σε μια μεταβλητή τύπου *Employee* μπορεί να βάλουμε την ημερομηνία γέννησης (*birthDate*) μεταγενέστερη της ημερομηνίας πρόσληψης (*emplDate*) ή αριθμό παιδιών (*numberOfChld*) αρνητικό.
- Σε μια μεταβλητή τύπου *GrElmn* μπορεί να βάλουμε αρνητικό ατομικό αριθμό (*geANumber*) ή αρνητικό ατομικό βάρος (*geAWeight*).

Εδώ θα μάθουμε ότι η C++, όπως και άλλες **αντικειμενοστρεφείς** γλώσσες προγραμματισμού, μας δίνουν τη δυνατότητα να δημιουργούμε αντικείμενα που έχουν πραγματική «κυριότητα του εαυτού τους» και δεν επιτρέπουν την «κακομεταχείρισή» τους από το περιβάλλον στο οποίο «ζουν».

19.1 Κλάσεις

Θα ξεκινήσουμε με τη διόρθωση των προβλημάτων της *Date*.[1] Πριν από οτιδήποτε άλλο θα πρέπει καταγράψουμε με ακρίβεια τη «συνθήκη νομιμότητας» για τη *Date*, δηλαδή τι είναι δεκτό και τι όχι:

$$(dYear > 0) \wedge (0 < dMonth \leq 12) \wedge (0 < dDay \leq lastDay(dYear, dMonth))$$

Αυτή είναι η **αναλλοίωτη της κλάσης** (class invariant) με την έννοια: ισχύει σε όλη τη διάρκεια της ζωής κάθε τιμής (αντικειμένου) τύπου *Date*.

Η *lastDay* είναι είναι μια συνάρτηση που, όταν την καλέσουμε όπως εδώ, μας δίνει την τελευταία ημέρα του μήνα *dMonth* του έτους *dYear*. Πώς θα είναι; Έτσι:

```
unsigned int lastDay( int y, int m )
{
   unsigned int fv;

   if ( m == 1 || m == 3 || m == 5 || m == 7 || m == 8 ||
        m == 10 || m == 12 )
      fv = 31;
   else if ( m == 4 || m == 6 || m == 9 || m == 11 )
      fv = 30;
   else // m == 2
      if ( isLeapYear(y) )  fv = 29;
                      else  fv = 28;
   return fv;
} // lastDay
```

[1] Αλλάζουμε και τα ονόματα των μελών: *dYear*, *dMonth*, *dDay* αντί για *year*, *month*, *day*. Η εξήγηση παρακάτω.

Και εκείνο το "`isLeapYear(y)`" τι είναι; Κλήση προς μια άλλη συνάρτηση

```
bool isLeapYear( int y )
{
   bool fv( false );

   if ( y%400 == 0 )                    fv = true;
   else if ( y%4 == 0 && y%100 != 0 )  fv = true;
   return fv;
} // isLeapYear
```

που τροφοδοτείται με ένα έτος *y* και μας επιστρέφει `true`, αν το έτος είναι δίσεκτο, αλλιώς `false`. Θα μπορούσαμε βέβαια να τη γράψουμε πιο απλά[2].

Αυτές οι συναρτήσεις φαίνονται ελλιπείς: Ούτε ένας έλεγχος για τις τιμές των παραμέτρων; Μπορούμε να βάλουμε ό,τι νάναι; Σωστή η παρατήρηση, αλλά… διάβασε παρακάτω.

Όπως είπαμε, για κάθε τιμή τύπου *Date* ή –όπως θα λέμε από εδώ και πέρα– *για κάθε αντικείμενο κλάσης Date, θα πρέπει να ισχύει η αναλλοίωτη*. Αυτό μπορεί να διασφαλίσθεί με *ελεγχόμενη* πρόσβαση στα μέλη των στοιχείων κλάσης *Date*. Δες πώς μπορεί να γίνει κάτι τέτοιο:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 )
   {  year = ay;  month = am;  day = ad;  }
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;
}; // Date
```

Πρόσεξε το "`private:`" πριν από τις δηλώσεις των μελών· σημαίνει ότι ένα πρόγραμμα που χρησιμοποιεί αντικείμενα κλάσης *Date* δεν έχει άμεση πρόσβαση στα μέλη των αντικειμένων. Οτιδήποτε δηλώνεται ως `private` δεν είναι «ορατό» έξω από το αντικείμενο. Έτσι, οι εντολές "`d1.dYear = -5; d1.dMonth = 29;`" είναι παράνομες: *δεν περνούν από τον μεταγλωττιστή*. Λέμε ότι τα μέλη έγιναν **εσωτερικά** (private).

Δεν κάναμε εσωτερικό τον δημιουργό, τον αφήσαμε **ανοικτό** (public), ώστε να έχουμε τη δυνατότητα να δηλώνουμε αντικείμενα κλάσης *Date*. Ο δημιουργός –και οτιδήποτε δηλώνεται μέσα στην κλάση– έχει πρόσβαση και στα εσωτερικά μέλη και μέσω αυτού μπορούμε:

- να ορίσουμε τις τιμές των μελών ή
- να τις αλλάξουμε.

Πάντως, δεν λύσαμε ακόμη το πρόβλημά μας: *μπορούμε ακόμη να δώσουμε μη αποδεκτές τιμές*. Επιπλέον, τώρα έχει προκύψει και ένα άλλο πρόβλημα: *πώς θα χρησιμοποιήσουμε τις τιμές των μελών;*

Ας ξεκινήσουμε με τη βελτίωση του δημιουργού: θα τον εφοδιάσουμε με ελέγχους:

```
   Date( int ay=1, int am=1, int ad=1 )
   {
      if ( ay <= 0 )
         throw . . .;
   // ay > 0
      dYear = ay;
      if ( am <= 0 || 12 < am )
         throw . . .;
   // dYear > 0 && ( 0 < am && am <= 12 )
      dMonth = am;
```

[2] Να η απλή μορφή:

```
bool isLeapYear( int y )
{  return ( y%400 == 0 || (y%4 == 0 && y%100 != 0) );  } // isLeapYear
```

```
      if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
         throw . . .;
   // dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
   // ( 0 < ad && ad <= lastDay(dYear, dMonth) )
      dDay = ad;
   // dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
   // ( 0 < dDay && dDay <= lastDay(dYear, dMonth) )
   } // Date
```

Όπως φαίνεται, *όταν ο δημιουργός τελειώσει τη δουλειά του –αν την τελειώσει– θα έχει δημιουργήσει ένα αντικείμενο που συμμορφώνεται με την αναλλοίωτη της κλάσης*.

Εδώ να παρατηρήσουμε δύο πράγματα:

- Όταν καλείται η *lastDay()* είναι σίγουρο ότι οι τιμές των ορισμάτων είναι «νόμιμες». Έτσι, δεν χρειάζονται οι έλεγχοι μέσα στη συνάρτηση που λέγαμε ότι λείπουν. Και πώς σιγουρεύουμε ότι δεν θα κληθεί από κάπου αλλού με λάθος ορίσματα; Θα την κρύψουμε, μαζί με την *isLeapYear()*, στην περιοχή **private** της κλάσης:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

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

Τώρα, οι δύο συναρτήσεις είναι προσβάσιμες από οτιδήποτε υπάρχει μέσα στον ορισμό της κλάσης αλλά όχι έξω από αυτήν.

- Δεν γράψαμε τον τύπο της εξαίρεσης· θα πρέπει να περιμένουμε να δούμε τις κλάσεις εξαιρέσεων της εφαρμογής-πελάτη; Όχι! *Μαζί με κάθε κλάση θα ορίζουμε και την αντίστοιχη κλάση εξαιρέσεων*. Εδώ θα ορίσουμε προς το παρόν μια:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr };
   char funcName[100];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
   DateXptn( const char* mn, int ec,
             int ev1=0, int ev2=0, int ev3=0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = ev1;  errVal2 = ev2;  errVal3 = ev3;  }
}; // DateXptn
```

Αργότερα θα την αλλάξουμε κάπως.

- Αν δεν βάζαμε τα "`// ΟΠΩΣ ΠΑΡΑΠΑΝΩ`" ο ορισμός της κλάσης θα ήταν ήδη μεγάλος και δεν θα ήταν εύκολο να ψάχνουμε κάτι που θέλουμε. Η C++ μας δίνει την εξής δυνατότητα: να βάλουμε μόνο τις επικεφαλίδες των συναρτήσεων και να τις ορίσουμε ολόκληρες έξω από την κλάση. Δηλαδή:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
private:
   unsigned int  dYear;
   unsigned char dMonth;
```

```
   unsigned char dDay;

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

και

```
bool Date::isLeapYear( int y )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
unsigned int Date::lastDay( int y, int m )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Δηλαδή: εδώ ορίζουμε την *isLeapYear()* που δηλωσαμε στην κλάση (και ονοματοχώρο) *Date*. Τα ίδια κάνουμε και για τη *lastDay()*.

Τα ίδια ισχύουν και για τον δημιουργό αλλά εδώ δεν θα βάλουμε "`// ΟΠΩΣ ΠΑΡΑΠΑΝΩ`"· θα τον γράψουμε ολόκληρο:

```
Date::Date( int ay, int am, int ad )
{
   if ( ay <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, ay );
// ay > 0
   dYear = ay;
   if ( am <= 0 || 12 < am )
      throw DateXptn( "Date", DateXptn::monthErr, am );
// dYear > 0 && ( 0 < am && am <= 12 )
   dMonth = am;
   if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
      throw DateXptn( "Date", DateXptn::dayErr,
                      dYear, dMonth, ad );
// dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
// ( 0 < ad && ad <= lastDay(dYear, dMonth) )
   dDay = ad;
// dYear > 0 && ( 0 < dMonth && dMonth <= 12 ) &&
// ( 0 < dDay && dDay <= lastDay(dYear, dMonth) )
} // Date
```

Πρόσεξε τα εξής:

- Οι *προκαθορισμένες τιμές των παραμέτρων εμφανίζονται μόνο στη δήλωση του δημιουργού*, μέσα στον ορισμό της κλάσης *αλλά δεν εμφανίζονται ξανά στον ορισμό του δημιουργού*.
- Μέσα στο σώμα της συνάρτησης δεν χρειάζεται να βάλουμε "`Date::`" στα ονόματα των μελών. Αφού βάλαμε "`Date::Date`" στην επικεφαλίδα είμαστε μέσα στον ονοματοχώρο *Date* και το "`dYear`" σημαίνει "`Date::dYear`".
- Όταν έχουμε λάθος στο έτος ή στον μήνα στέλνουμε με την εξαίρεση μια τιμή αλλά όταν έχουμε λάθος στην ημέρα στέλνουμε τρεις. Γιατί;
  - Διότι το να πούμε ότι το *ad* είχε τιμή 31 δεν λέει και πολλά πράγματα αν δεν ξέρουμε ότι το *dMonth* έχει τιμή 6.
  - Και το να πούμε ότι το *ad* είχε τιμή 29 δεν δείχνει κάποιο πρόβλημα αν δεν ξέρουμε ότι το *dMonth* έχει τιμή 2 και το *dYear* έχει τιμή 2001.

Να δούμε τώρα το άλλο πρόβλημα: «*πώς θα χρησιμοποιήσουμε τις τιμές των μελών;*» Για την πρόσβαση στα μέλη εφοδιάζουμε την κλάση με κατάλληλες *ανοικτές* **συναρτήσεις-μέλη** (member fumctions) ή **μεθόδους** (methods):

```
struct Date
{
   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 );  }
private:
```

```
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

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

Οι *getYear()*, *getMonth()*, *getDay()* είναι τέτοιες μέθοδοι. Πώς τις χρησιμοποιούμε; Δες το πρόγραμμα:

```
   Date d1( 2010, 10, 13 );
   cout << "d1: "
        << d1.getDay() << '.' << d1.getMonth() << '.' << d1.getYear() << endl;

   d1 = Date( 2008, 7, 9 );
   cout << "d1: "
        << d1.getDay() << '.' << d1.getMonth() << '.' << d1.getYear() << endl;
```

θα δώσει:

```
d1: 13.10.2010
d1: 9.7.2008
```

Όπως βλέπεις, χρησιμοποιήσαμε τον δημιουργό όχι μόνον για να δηλώσουμε τη *d1* με αρχική τιμή, αλλά και για να αλλάξουμε την τιμή της. Στην §15.3 μάθαμε ότι με τη "`Date(2008, 7, 9)`" καλούμε τον δημιουργό για να δημιουργήσει μια τιμή κλάσης *Date*. Αυτήν την τιμή εκχωρούμε στη *d1*.

Πρόσεξε όμως τη χρήση των μεθόδων: γράφουμε

"`d1.getDay()`", "`d1.getMonth()`", "`d1.getYear()`"

και όχι

"`Date::getDay(d1)`", "`Date::getMonth(d1)`", "`Date::getYear(d1)`"

Δηλαδή *οι μέθοδοι ανήκουν στο αντικείμενο και όχι στην κλάση!* Δηλαδή:

- ***Ένα αντικείμενο περικλείει –εκτός από δεδομένα– και τις μεθόδους για τον χειρισμό τους.***

Η **περίκλειση** (encapsulation) δεδομένων και εργαλείων χειρισμού τους είναι βασικό χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού.

Έχουμε ήδη χρησιμοποιήσει κατά κόρον την κλήση μεθόδων αντικειμένου με τον τελεστή "`.`"· θυμίσου τα `s.length()` ή `s.c_str()` για ένα αντικείμενο *s* κλάσης (*std*`::`)*string* ή τα `bin.open(…)` ή `bin.read(…)` για ένα αντικείμενο *bin* κλάσης (*std*`::`)*ifstream*. Μην αμφιβάλλεις ότι παρομοίως χρησιμοποιούμε και τον "`->`". Αν έχουμε:

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

η

```
   cout << "*pd: "
        << pd->getDay() << '.' << pd->getMonth() << '.'
        << pd->getYear() << endl;
```

θα δώσει:

```
*pd: 9.7.2008
```

Είδαμε πιο πάνω ότι με χρήση δημιουργού (και της εκχώρησης) μπορούμε να αλλάξουμε την τιμή ενός αντικειμένου. Αν όμως θέλουμε να αλλάξουμε, ας πούμε, μόνο τον μήνα τι κάνουμε; Να ένα παράδειγμα:

```
   d1 = Date( d1.getYear(), 5, d1.getDay() );
```

Με αυτήν την εντολή βάζουμε τον μήνα "5" κρατώντας χωρίς αλλαγή έτος και ημέρα.

Καλό θα είναι όμως να μπορούμε να αλλάξουμε την τιμή οποιουδήποτε μέλους ενός αντικειμένου. Δηλώνουμε λοιπόν τρεις νέες μεθόδους:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
```

```
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth() const {  return dMonth;  }
   unsigned int getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
private:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

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

και τις ορίζουμε ως εξής:

```
void Date::setYear( int ay )
{
   if ( ay <= 0 )
      throw DateXptn( "setYear", DateXptn::yearErr, ay );
   if ( dDay <= 0 || lastDay(ay, dMonth) < dDay )
      throw DateXptn( "setYear", DateXptn::dayErr,
                      ay, dMonth, dDay );
   dYear = ay;
} // Date::setYear
```

Καλά ο πρώτος έλεγχος, ο δεύτερος τι είναι; Σκέψου την περίπτωση που η *d* έχει ως τιμή την **29.02.2000** και κάνουμε απόπειρα (**d.setYear(2001)**) να αλλάξουμε το έτος σε **2001**. Παρόμοιο πρόβλημα υπάρχει και στην

```
void Date::setMonth( int am )
{
   if ( am <= 0 || 12 < am )
      throw DateXptn( "setMonth", DateXptn::monthErr, am );
   if ( dDay <= 0 || lastDay(dYear, am) < dDay )
      throw DateXptn( "setMonth", DateXptn::dayErr,
                      dYear, am, dDay );
   dMonth = am;
} // Date::setMonth
```

Στην περίπτωση αυτή ο δεύτερος έλεγχος χρειάζεται αν –για παράδειγμα– έχουμε τιμή της *d* την **31.12.2010** και δώσουμε **d.setMonth(6)**.

```
void Date::setDay( int ad )
{
   if ( ad <= 0 || lastDay(dYear, dMonth) < ad )
      throw DateXptn( "setDay", DateXptn::dayErr,
                      dYear, dMonth, ad );
   dDay = ad;
} // Date::setDay
```

Η *setDay* είναι απλή και δεν χρειάζεται εξηγήσεις.

Σημείωση: ▶

Οι τρεις συναρτήσεις "*set*" που γράψαμε είναι καλές για επίδειξη αλλά, για την κλάση *Date*, σχεδόν άχρηστες σε πραγματικό πρόγραμμα. Για σκέψου τι θα γίνεται όποτε έχεις να αλλάξεις τις τιμές δύο μελών: Για να μην πέσει εξαίρεση θα πρέπει να αλλάξεις πρώτα τη μέρα ή τον μήνα (ή το έτος). Πρόσεξε το εξής παράδειγμα: Αν η *d1* έχει τιμή 31.05.2007 και θέλουμε να την αλλάξουμε σε 30.04.2007 θα πρέπει να αλλάξουμε πρώτα τη μέρα και μετά τον μήνα. Αν θέλουμε να αλλάξουμε την 30.04.2007 σε 31.05.2007 θα πρέπει να αλλάξουμε πρώτα τον μήνα. Συμπέρασμα: στη *Date* δεν βγαίνει άκρη με τις "*set*". *Η αλλαγή τιμής με τη χρήση του δημιουργού είναι η πιο σίγουρη.*

```
   Date d1( 2007, 5, 31 );                      // d1 == 31.05.2007

   d1 = Date( d1.getYear(), 4, 30 );            // d1 == 30.04.2007
   d1 = Date( d1.getYear(), 5, 31 );            // d1 == 31.05.2007
   d1 = Date( d1.getYear(), 12, d1.getDay() ); // d1 == 31.12.2007
```

```
   d1 = Date( d1.getYear()+1,
              d1.getMonth(), d1.getDay() );    // d1 == 31.12.2008
```

Με αυτόν τον τρόπο κάνει περισσότερες πράξεις αλλά είναι αυτός που δουλεύει σίγουρα. ◀

Για να λύσεις την άσκ. 15-1 θα πρέπει να έγραψες για τη *Date* μια:

```
void Date_load( Date& a, istream& bin )
{
   bin.read( reinterpret_cast<char*>(&a.year), sizeof(a.year) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&a.month),
                sizeof(a.month) );
      bin.read( reinterpret_cast<char*>(&a.day), sizeof(a.day) );
      if ( bin.fail() )
         throw XXX_Xptn( "Date_load", XXX_Xptn::cannotRead );
   }
}; // Date_load
```

Πώς θα τη μετατρέψουμε ώστε να την περικλείσουμε στα αντικείμενα κλάσης *Date;* Σε αυτήν τη μορφή τη χρησιμοποιούσαμε ως εξής: Αν

```
   ifstream dateFile( ..., ios_base::binary );
   Date d;
```

γράφουμε:

```
   Date_load( d, dateFile );
```

Τώρα, η *load()* θα είναι «ιδιοκτησία» του αντικειμένου και θα γράφουμε:

```
   d.load( dateFile );
```

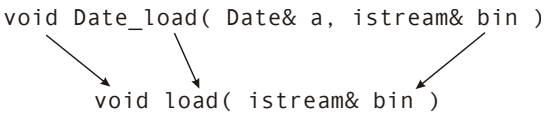
Θα πρέπει λοιπόν να τη δηλώσουμε μέσα στην κλάση:

```
struct Date
{
// . . .
   void load( istream& bin );
// . . .
private:
// . . .
}; // Date
```

και να την ορίσουμε:

```
void Date::load( istream& bin )
{
   bin.read( reinterpret_cast<char*>(&dYear), sizeof(dYear) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&dMonth),
                sizeof(dMonth) );
      bin.read( reinterpret_cast<char*>(&dDay), sizeof(dDay) );
      if ( bin.fail() )
         throw DateXptn( "load", DateXptn::cannotRead );
   }
}; // Date::load
```

Τη σχέση μεταξύ της εξωτερικής συνάρτησης και της περικλειόμενης μεθόδου τη δίνουμε διαγραμματικώς κάπως έτσι:



Δεν υπάρχει αλλαγή στον τύπο (‘**void**” στην περίπτωσή μας) και στις παραμέτρους εκτός από την παράμετρο-αντικείμενο. Η παράμετρος-αντικείμενο εξαφανίζεται αφού τώρα το αντικείμενο είναι ο «ιδιοκτήτης» της μεθόδου.

19.1.1 "const"

Στην άσκ. 15-1 δίνεται και η συνάρτηση

```
void Date_save( const Date& a, ostream& bout )
{
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&a.year),
               sizeof(a.year) );
   bout.write( reinterpret_cast<const char*>(&a.month),
               sizeof(a.month) );
   bout.write( reinterpret_cast<const char*>(&a.day),
               sizeof(a.day) );
   if ( bout.fail() )
      throw XXX_Xptn( "Date_save", XXX_Xptn::cannotWrite );
}; // Date_save
```

που τη μετατρέπουμε σε μέθοδο ως εξής: Τη δηλώνουμε μέσα στην κλάση:

```
// . . .
   void save( istream& bin ) const;
// . . .
```

και την ορίζουμε:

```
void Date::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw DateXptn( "save", DateXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(dYear),
               sizeof(dYear) );
   bout.write( reinterpret_cast<const char*>(dMonth),
               sizeof(dMonth) );
   bout.write( reinterpret_cast<const char*>(dDay),
               sizeof(dDay) );
   if ( bout.fail() )
      throw DateXptn( "save", DateXptn::cannotWrite );
}; // Date::save
```

Το νέο στοιχείο εδώ είναι

- το "`const`" στον τύπο του αντικειμένου στην *Date_save()*
- που εμφανίζεται στο τέλος της επικεφαλίδας της μεθόδου *Date::save()*.

Τι σημαίνει; Στη *Date_save()* εγγυάται ότι η συνάρτηση δεν αλλάζει την τιμή του αντικειμένου *a*. Το ίδιο σημαίνει και στη μέθοδο:

♦ ***Με το χαρακτηριστικό "const" στο τέλος της επικεφαλίδας της μεθόδου σημειώνουμε ότι η μέθοδος δεν αλλάζει την τιμή οποιουδήποτε μέλους του αντικειμένου που την περικλείει.***

Τώρα οι αντιστοιχίες φαίνονται διαγραμματικώς ως εξής:

```
void Date_save( const Date& a, ostream& bout )

void save( ostream& bout ) const
```

Πρόσεξε ότι βλέπουμε το "`const`" και στις τρεις μεθόδους "*get*" αλλά όχι στις "*set*" ούτε στη *load()*.

19.1.2 Βοηθητικές Συναρτήσεις

Βάλαμε τις συναρτήσεις *lastDay()* και *isLeapYear()* στην περιοχή `private` της κλάσης ώστε να είναι διαθέσιμες μόνο στις μεθόδους της. Από αυτές καλούμε τη *lastDay* και μάλιστα, πολύ προσεκτικά, με «σίγουρες» τιμές ορισμάτων!

Ακόμη, πρόσεξε ότι οι δύο συναρτήσεις δεν αναφέρονται σε συγκεκριμένο αντικείμενο· αναφέρονται στην κλάση. Λέμε ότι είναι **βοηθητικές συναρτήσεις** (helper functions) της κλάσης.[3]

Για να το καταλάβεις αυτό το «*δεν αναφέρονται σε συγκεκριμένο αντικείμενο*», δες πώς θα ήταν αν αναφέρονταν:

```
bool isLeapYear() const
{
   bool fv( false );

   if ( dYear%400 == 0 )                        fv = true;
   else if ( dYear%4 == 0 && dYear%100 != 0 )  fv = true;
   return fv;
} // isLeapYear
```

Δηλαδή, έχουμε μια συνάρτηση χωρίς παραμέτρους που υπολογίζει την τιμή της με βάση τις τιμές των μελών του αντικειμένου. Το ίδιο και η

```
unsigned int lastDay() const
{
   unsigned int fv;

   if ( dMonth == 1 || dMonth == 3 || dMonth == 5 || dMonth == 7
        || dMonth == 8 || dMonth == 10 || dMonth == 12 )
      fv = 31;
   else if ( dMonth == 4 || dMonth == 6 || dMonth == 9 ||
             dMonth == 11 )
      fv = 30;
   else // month == 2
      if ( isLeapYear(dYear) )  fv = 29;
                          else  fv = 28;
   return fv;
} // lastDay
```

Αν όμως γυρίσεις πίσω και δεις ότι καλούμε τη *lastDay()* δύο φορές με παραμέτρους *dMonth* και *dYear* και άλλες δύο φορές με άλλες παραμέτρους. Καταλαβαίνεις ότι μάλλον θα περιπλέκαμε τα προγράμματά μας.

Το σοβαρότερο πρόβλημα όμως είναι το νοηματικό. Αν έχουμε

```
Date d;
```

τι νόημα έχει το `d.lastDay();` Τελευταία ημέρα του μήνα που υπάρχει στην τιμή της *d*!!! Και το `d.isLeapYear()` θα μας δώσει `true` αν είναι δίσεκτη η ημερομηνία *d*;! Όχι βέβαια, αφορά μόνον το έτος της ημερομηνίας. Καταλαβαίνεις ότι αυτά δεν στέκουν.

19.1.3 "class" και "public"

Η κλάση μας τώρα έχει γίνει έτσι:

```
struct Date
{
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned char getMonth() const {  return dMonth;  }
   unsigned char getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   void load( istream& bin );
   void save( ostream& bout ) const;
private:
   unsigned int  dYear;
   unsigned char dMonth;
```

[3] Αργότερα θα δεις ότι θα τις διακοσμήσουμε και με ένα "`static`"!

```
   unsigned char dDay;

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

Ένας άλλος τρόπος να τη γράψουμε είναι ο εξής:

```
class Date
{
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   bool isLeapYear( int y );
   unsigned int lastDay( int y, int m );
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned char getMonth() const {  return dMonth;  }
   unsigned char getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   void load( istream& bin );
   void save( ostream& bout ) const;
}; // Date
```

Δηλώνοντας μια κλάση με το "`struct`" εννοούμε ότι *κατ' αρχήν έχει όλα τα μέλη της ανοικτά*. Αν θέλουμε να κρύψουμε κάποια από αυτά θα πρέπει να τα βάλουμε σε περιοχή "`private`".

Αν η δήλωση γίνεται με το "`class`" η κλάση *κατ' αρχήν έχει όλα τα μέλη της εσωτερικά*. Αν θέλουμε να ανοίξουμε κάποια από αυτά θα πρέπει να τα βάλουμε σε περιοχή "`public`".

Με οποιονδήποτε από τους δύο τρόπους και αν κάνουμε τη δήλωση:

- Έχουμε τη δυνατότητα για πολλές περιοχές "`public`" και "`private`".
- Η οποιαδήποτε επαφή του κάθε προγράμματος με το αντικείμενο γίνεται μέσω αυτών που υπάρχουν στις περιοχές "`public`". Αυτά αποτελούν και το **τμήμα διεπαφής** (interface part) του αντικειμένου.

Πάντως παρακάτω θα δώσουμε ένα κριτήριο για να επιλέγεις μεταξύ "`class`" και "`struct`".

Τη συγκεκριμένη κλάση θα την ορίζουμε ως εξής:

```
class Date
{
public:
   Date( int ay=1, int am=1, int ad=1 );
   unsigned int getYear() const {  return dYear;  }
   unsigned int getMonth() const {  return dMonth;  }
   unsigned int getDay() const {  return dDay;  }
   void setYear( int ay );
   void setMonth( int am );
   void setDay( int ad );
   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
```

Οι τρεις τρόποι ορισμού της *Date* που είδες παραπάνω είναι ισοδύναμοι.

Η κλάση εξαιρέσεων είναι:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[100];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
   DateXptn( const char* mn, int ec,
             int ev1 = 0, int ev2 = 0, int ev3 = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errVal1 = ev1;  errVal2 = ev2;  errVal3 = ev3;  }
}; // DateXptn
```

19.1.4 Επιφόρτωση Τελεστών

Για τη *Date* είχαμε επιφορτώσει και κάποιους τελεστές (§15.5). Τι θα γίνει με αυτούς; Θα τους επιφορτώσουμε με μεθόδους;

Για τους συγκεκριμένους ("`<<`", "`<`", "`==`") κρατούμε τις υλοποιήσεις όπως τις έχουμε αλλά με μια διαφορά: αφού είναι εξωτερικές για την κλάση δεν είναι δυνατόν να έχουν πρόσβαση στα μέλη του αντικειμένου και πρέπει να χρησιμοποιήσουμε τις αντίστοιχες μεθόδους "*get*".

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << rhs.getDay() << '.' << rhs.getMonth() << '.'
               << rhs.getYear();
} // operator<<( ostream& tout, const Date

bool operator<( const Date& lhs, const Date& rhs )
{
   bool fv;

   if ( lhs.getYear() < rhs.getYear() )           fv = true;
   else if ( lhs.getYear() > rhs.getYear() )      fv = false;
   else // lhs.getYear() == rhs.getYear()
   {
      if ( lhs.getMonth() < rhs.getMonth() )      fv = true;
      else if ( lhs.getMonth() > rhs.getMonth() ) fv = false;
      else // lhs.getYear() == rhs.getMonth() &&
           // lhs.getMonth() == rhs.getMonth()
         fv = ( lhs.getDay() < rhs.getDay() );
   }
   return fv;
} // operator<( const Date . . .

bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.getYear() == rhs.getYear() &&
            lhs.getMonth() == rhs.getMonth() &&
            lhs.getDay() == rhs.getDay() );
}; // operator==( const Date
```

Θα πεις «ε, όχι και "*όπως είναι αλλά με μια διαφορά*"! Η διαφορά είναι σοβαρή αφού φορτώνουμε τις συναρτήσεις με τόσες κλήσεις μεθόδων, δηλαδή άλλων συναρτήσεων». Ναι μεν αλλά, όπως θα δεις στη συνέχεια, αυτές είναι "`inline`".

Πάντως, στη συνέχεια θα δούμε έναν τρόπο για να δίνουμε δικαίωμα σε μια (εξωτερική) συνάρτηση να έχει κατ' ευθείαν πρόσβαση στα μέλη αντικειμένου που βρίσκονται σε περιοχή **private**.

19.1.5 Ονοματολογία

Γιατί αλλάξαμε τα ονόματα των μελών της *Date* σε *dYear*, *dMonth* και *dDay*; Αυτή είναι μια συνηθισμένη σύμβαση για να φαίνεται ότι πρόκειται για μέλη μιας κλάσης (που το όνομά της αρχίζει από *d*). Μπορεί να δεις και δύο ή τρία γράμματα (στην αρχή).

Άλλοι προτιμούν να βάζουν πριν από το όνομα του μέλους το «*m*» –από το «member» (μέλος)– και στην περίπτωσή μας θα έγραφαν: *mYear*, *mMonth* και *mDay*. Άλλοι πάλι θα έγραφαν *_year*, *_month* και *_day*.

19.2 Το Είδος των Μεθόδων

Οι κανόνες που έχουμε μάθει για το είδος των συναρτήσεων ισχύουν και για τις μεθόδους μιας κλάσης. Αλλά για να τους εφαρμόσεις θα πρέπει να παίρνεις υπόψη σου ότι *και το αντικείμενο παίζει ρόλο ορίσματος* όπως είδαμε στο τέλος της §19.1 και στην §19.1.1.

Ας δούμε τη *setYear()*. Αν τη γράφαμε για τη «παλιά» "`struct Date`" πώς θα ήταν; Κάπως έτσι:

```
void Date_setYear( Date& d, int ay )
```

Εδώ, σε συμφωνία με τους κανόνες μας, βάζουμε "`void`" διότι η συνάρτηση αλλάζει την τιμή της πρώτης παραμέτρου. Με τις αντιστοιχίες που είδαμε στην §19.1 γράφουμε μέθοδο:

```
void setYear( int ay )
```

Η *getYear()* για τη «παλιά» "`struct Date`" θα ήταν κάπως έτσι:

```
unsigned int Date_getYear( Date d )
```

ή, καλύτερα,

```
unsigned int Date_getYear( const Date& d )
```

Σε συμφωνία με τους κανόνες μας, βάζουμε γράφουμε συνάρτηση με τύπο διότι το μόνο που έχει να κάνει είναι να επιστρέψει μια τιμή. Με τις αντιστοιχίες που είδαμε στην §19.1.1 γράφουμε μέθοδο:

```
unsigned int getYear() const
```

Δηλαδή: *Για να επιλέξουμε το είδος μιας μεθόδου για μια κλάση K βάζουμε στη μέθοδο μια επιπλέον παράμετρο* –`K& a` *ή* `const K& a`– *και παίρνουμε την απόφασή μας.*

Όπως καταλαβαίνεις, αν γράφεις τις μεθόδους σύμφωνα με τους κανόνες που έχουμε βάλει, όποτε γράφεις μέθοδο με τύπο θα πρέπει να της δίνεις και το χαρακτηριστικό "`const`".

Πάντως και τώρα, για να είμαστε σε συμφωνία με τη «φιλοσοφία της C++» (C), σε ορισμένες περιπτώσεις θα παραβιάζουμε τους κανόνες μας.

19.3 Κατανομή σε Αρχεία

Ο συνηθισμένος τρόπος κατανομής του ορισμού της κλάσης σε αρχεία είναι ο εξής –για την *Date*:

- Στο αρχείο `Date.h` (μετά και την αλλαγή των ονομάτων):

```
#ifndef _DATE_H
#define _DATE_H

#include <fstream>
#include <string>

using namespace std;

class Date
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

```
struct DateXptn
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

ostream& operator<<( ostream& tout, const Date& rhs );
bool operator<( const Date& lhs, const Date& rhs );
bool operator==( const Date& lhs, const Date& rhs );

#endif // _DATE_H
```

- Στο αρχείο `Date.cpp`:

```
#ifndef _DATE_CPP
#define _DATE_CPP

#include <fstream>
#include "Date.h"

Date::Date( int ay, int am, int ad )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setYear( int ay )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setMonth( int am )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::setDay( int ad )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::load( istream& bin )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Date::save( ostream& bout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool Date::isLeapYear( int y )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
unsigned int Date::lastDay( int y, int m )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

ostream& operator<<( ostream& tout, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool operator<( const Date& lhs, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
bool operator==( const Date& lhs, const Date& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _DATE_CPP
```

Τώρα, μέσα στο αρχείο του προγράμματος που χρησιμοποιεί τη *Date*, αρκεί να βάλεις "`#include "Date.cpp"`".

Αν θέλεις μπορείς να κάνεις και χωριστή μεταγλώττιση και να δώσεις στο project του προγράμματός σου για σύνδεση το `Date.o` (ή `Date.obj`). Στην περίπτωση αυτή στο πρόγραμμά σου θα βάλεις "`#include "Date.h"`".[4] Θα δούμε τέτοιο παράδειγμα αργότερα.

19.3.1 Τα «Μυστικά» της Υλοποίησης

Σε προηγούμενη παράγραφο λέγαμε για τις βοηθητικές συναρτήσεις: «*Βάλαμε τις συναρτήσεις lastDay και isLeapYear στην περιοχή* `private` *της κλάσης ώστε να είναι διαθέσιμες μόνο στις μεθόδους της.*» Και στην υποσημείωση, πιο πάνω: «*Στον "πελάτη", για τον οποίον συζητούσαμε στο προηγούμενο κεφάλαιο, θα δώσεις τα* `Date.h` *και* `Date.obj`.» Μα τι γίνεται εδώ; Όλα τα κρύβουμε, όλα είναι μυστικά; Ναι! Και μάλιστα με δύο έννοιες:

- **Απόκρυψη πληροφορίας** (information hiding): Το πώς υλοποιείται η κάθε μέθοδος δεν αφορά αυτόν που ζητάει κάτι από ένα αντικείμενο. Του λέει «θέλω να γίνει αυτό» ή «θέλω αυτήν την πληροφορία» και –αν δεν ζητείται κάτι παράνομο, δηλαδή παραβίαση

[4] Στον "πελάτη", για τον οποίον συζητούσαμε στο προηγούμενο κεφάλαιο, θα δώσεις τα `Date.h` και `Date.obj`.

της αναλλοίωτης– το αντικείμενο αποκρίνεται με την αντίστοιχη μέθοδο. Αν αυτός που διατυπώνει την απαίτηση (ένα πρόγραμμα-πελάτης ή ένα άλλο αντικείμενο) δεν γνωρίζει τον –για την ακρίβεια: δεν στηρίζεται στον– τρόπο υλοποίησης της μεθόδου τότε έχουμε δυνατότητα να την αλλάξουμε – συνήθως να τη βελτιώσουμε– *χωρίς να χρειαστεί να μεταγλωττίσουμε ολόκληρη την εφαρμογή*: μεταγλωττίζουμε μόνον την υλοποίηση της κλάσης και συνδέουμε το νέο αρχείο obj που προκύπτει με την εφαρμογή. Άλλωστε η πείρα λέει ότι όταν οι προγραμματιστές χρησιμοποιούν μια κλάση βασιζόμενοι μόνο στις προδιαγραφές των μεθόδων –και χωρίς να ξέρουν την υλοποίηση– γράφουν προγράμματα καλύτερης ποιότητας (χωρίς «εξυπνάδες»!)

- **Εμπορικό μυστικό υλοποίησης**: Όταν ένας προγραμματιστής γράφει μια κλάση για έναν πελάτη μπορεί να του δώσει
  - τον ορισμό της κλάσης και το αρχείο obj ή
  - όλον τον αρχικό κώδικα, αλλά σε πολύ υψηλότερη τιμή.

  Στην πρώτη περίπτωση ο προγραμματιστής έχει δικαίωμα να κρατήσει μυστικές από τον πελάτη τις λεπτομέρειες της υλοποίησης.

19.4 Μέθοδοι "inline"

Στην *Date*, όπως την είδαμε πιο πάνω, πρόσεξε και κάτι άλλο: Όλο το τμήμα διεπαφής περιέχει δηλώσεις εκτός από τις τρεις (απλούστερες) μεθόδους, τις "*get*" για τις οποίες δίνουμε πλήρεις ορισμούς. Κάτι τέτοιο θα το βλέπεις πολύ συχνά. Γιατί;

♦ ***Όποια μέθοδος ορίζεται μέσα στην κλάση θεωρείται ότι έχει και το χαρακτηριστικό "`inline`" παρ' όλο που δεν το γράφουμε.***

Οι τρεις μέθοδοι "*get*" είναι αρκετά απλές και το "`inline`" θα έχει αποτέλεσμα. Αν θέλεις να τις βγάλεις έξω από τον ορισμό της κλάσης και να μην χάσεις το πλεονέκτημα τις δηλώνεις ως:

```
class Date
// . . .
   unsigned int getYear() const;
   unsigned char getMonth() const;
   unsigned char getDay() const;
// . . .
}; // Date
```

και τις ορίζεις:

```
inline unsigned int Date::getYear() const {  return dYear;  }
inline unsigned int getMonth()
                       const {  return static_cast<unsigned int>( dMonth );  }
inline unsigned int getDay() const
                              {  return static_cast<unsigned int>( dDay );  }
```

Καταλαβαίνεις τώρα γιατί η επιφόρτωση των τελεστών δεν θα επιβαρύνει τον χρόνο εκτέλεσης του προγράμματος παρά το ότι καλούμε μεθόδους "*get*".

Φυσικά, μπορείς να βάλεις το "`inline`" και σε οποιαδήποτε άλλη μέθοδο και να αφήσεις τον μεταγλωττιστή να αποφασίσει τι γίνεται και τι δεν γίνεται.

19.5 Αναλλοίωτη της Κλάσης – Κλάσεις Εξαιρέσεων

Όπως είπαμε, για κάθε αντικείμενο μιας κλάσης θα πρέπει πάντοτε να ισχύει η **αναλλοίωτη της κλάσης** (class invariant). Η συνθήκη

$$(dYear > 0) \wedge (0 < dMonth \leq 12) \wedge (0 < dDay \leq lastDay(dYear, dMonth))$$

είναι η αναλλοίωτη της κλάσης *Date*.

Η αναλλοίωτη μπορεί να παραβιάζεται προσωρινώς, όταν εκτελείται κάποια μέθοδος, αλλά ισχύει πριν την έναρξη και μετά τον τερματισμό της εκτέλεσής της.

Για κάθε κλάση που θα γράφουμε θα γράφουμε και μιαν αντίστοιχη κλάση (δομή) εξαιρέσεων σαν και αυτές που γράφαμε μέχρι τώρα. *Η αναλλοίωτη είναι ο οδηγός μας για το τι εξαιρέσεις (με ποιον κωδικό σφάλματος) ρίχνουμε και από πού.*

♦ ***Κάθε φορά που οδηγούμαστε σε παραβίαση της αναλλοίωτης ρίχνουμε εξαίρεση.***

Στη *DateXptn* οι κωδικοί σφάλματος αντιστοιχούν σε παραβιάσεις τμημάτων της αναλλοίωτης:

- Ο *yearErr* αντιστοιχεί στην παραβίαση της *dYear* > 0.
- Ο *monthErr* στην παραβίαση της 0 < *dMonth* ≤ 12 και
- ο *dayErr* στην παραβίαση της 0 < *dDay* ≤ *lastDay*(*dYear*, *dMonth*).

Εξαιρέσεις ρίχνουμε και όταν διαγνώσουμε πρόβλημα στην αλληλεπίδραση ενός αντικειμένου με το περιβάλλον του. Με άλλα λόγια: αφού η αλληλεπίδραση με το περιβάλλον γίνεται με τις μεθόδους ρίχνουμε εξαίρεση όταν κατά την εκτέλεση κάποιας μεθόδου παραβιάζονται οι προδιαγραφές της.

Για παράδειγμα, ας πάρουμε τη *Date* και την *DateXptn*. Ποιες είναι οι προδιαγραφές της *load();*

Προϋπόθεση: Το ρεύμα *bin* δεν έχει πρόβλημα εκτός από *eof*.

Απαίτηση: Στο αντικείμενο (ιδιοκτήτη της *load()*) φορτώνεται η τιμή του επόμενου αντικειμένου από το αρχείο.

Κανονικά θε πρέπει να γράψουμε:

```
void Date::load( istream& bin )
{
   if ( bin.fail() && !bin.eof() )
         throw DateXptn( "load", DateXptn::fileNotOpen );
   bin.read( reinterpret_cast<char*>(&dYear), sizeof(dYear) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&dMonth),
                sizeof(dMonth) );
      bin.read( reinterpret_cast<char*>(&dDay), sizeof(dDay) );
      if ( bin.fail() )
         throw DateXptn( "load", DateXptn::cannotRead );
   }
}; // Date::load
```

- Αν η *αρχική* `bin.fail()` επιστρέψει `true` παραβιάζεται η προϋπόθεση και ρίχνουμε εξαίρεση *DateXptn* με κωδικό σφάλματος *fileNotOpen* (δεν είναι και τόσο κυριολεκτικός).
- Αν επιστρέψει `true` η *τελική* `bin.fail()` σημαίνει ότι απέτυχε η ανάγνωση (ολικώς ή μερικώς) και έτσι δεν θα υπάρξει ανταπόκριση στη απαίτηση. Για τον λόγο αυτόν ρίχνουμε εξαίρεση *DateXptn* με κωδικό *cannotRead*.

Παρατηρήσεις: ▶

1. Γιατί χειριζόμαστε ξεχωριστά την περίπτωση *eof*; Διότι, όπως έχεις δει ήδη, το να διαβάζουμε μέχρι το τέλος αρχείου είναι πολύ συνηθισμένο.

Στη συνέχεια, όπως θα δεις, δεν θα βάζουμε τον πρώτο έλεγχο `bin.fail()`. Θα χρησιμοποιούμε την πρώτη μορφή της *load()*.

2. Πάντως η *load()* χρειάζεται περισσότερη προσοχή. Για παράδειγμα, αν αποτύχει το διάβασμα, θα πρέπει να μην καταστρέφει την τιμή του αντικειμένου. Θα την ξαναδούμε στη συνέχεια.◀

Πρόσεξε ακόμη κάτι πολύ σημαντικό που προκύπτει από τη διαχείριση με μεθόδους μελών που έχουμε περικλείσει σε περιοχές **private**: Αν δεις ότι σε κάποιο μέλος περνάνε απαράδεκτες τιμές ξέρεις πού ακριβώς –σε ποιες μεθόδους– πρέπει να ψάξεις για το πρόβλημα. Για παράδειγμα, αν δεις σε κάποιο πρόγραμμά σου ότι σε αντικείμενα κλάσης *Date* το *dYear*

παίρνει απαράδεκτες τιμές θα πρέπει να ξαναδείς τον *δημιουργό* και τη *setYear* αφού μόνο εκεί ορίζεται ή αλλάζει η τιμή αυτού του μέλους.

Τώρα θα συζητήσουμε δύο προβλήματα που έχουμε με τις κλάσεις εξαιρέσεων όπως τις έχουμε τυποποιήσει.

Το *πρώτο πρόβλημα* έχει σχέση με τα αντικείμενα: Η εξαίρεση θα μας φέρει τη μέθοδο που εμφανίστηκε το πρόβλημα, το είδος του προβλήματος και τις τιμές που το προκάλεσαν –π.χ. «*αποπειράθηκες να δώσεις με τη setYear αρνητική τιμή* (–7) *στο dYear*»– αλλά δεν θα μας πει σε ποιο αντικείμενο έγιναν όλα αυτά. Σε ένα δοκιμαστικό προγραμματάκι που έχει 3 ή 4 αντικείμενα δεν υπάρχει πρόβλημα. Αν όμως, σε κάποιο πρόγραμμα, έχουμε 4793 αντικείμενα τύπου *Date* αυτή η πληροφορία δεν μας λέει και πολλά. Μια πρώτη σκέψη είναι να αντιγράφουμε στην εξαίρεση και το *κλειδί* (§15.5.1) του αντικειμένου που έχει το πρόβλημα (τέτοιο παράδειγμα θα δούμε αργότερα). Αλλά πολύ συχνά αυτό δεν είναι αρκετό. Σκέψου την περίπτωση που τα 4780 αντικείμενα από αυτά που λέγαμε παραπάνω, ανήκουν σε 2390 στοιχεία ενός πίνακα κλάσης *Employee* (δύο ημερομηνίες στο καθένα, §15.1). Προφανώς δεν μπορούμε να αποκλείσουμε την εμφάνιση της ίδιας ημερομηνίας περισσότερες από μία φορές. Αν εξοπλίσουμε κάθε αντικείμενο με ένα επιπλέον μέλος, μια **ταυτότητα αντικειμένου** (object identifier) που θα την αντιγράφουμε και στο αντικείμενο της εξαίρεσης λύνουμε το πρόβλημά μας.[5]

Το *δεύτερο* πρόβλημα μπορεί να το έχεις αντιμετωπίσει ήδη αν έγραψες κάπως μεγάλα προγράμματα: «Ναι, η εξαίρεση ρίχτηκε από τη συνάρτηση *myFunc*, αλλά η *myFunc* καλείται στο πρόγραμμά μου σε 37 διαφορετικές «διαδρομές» εκτέλεσης!» Αργότερα θα δώσουμε μια τεχνική για να μπορείς να έχεις ακριβέστερη πληροφορία.

Και στα δύο προβλήματα μπορεί να πάρεις σημαντική βοηθεια, στη φάση των δοκιμών, από ένα καλό **πρόγραμμα διόρθωσης λαθών** (debugger). Αλλά αν το πρόβλημα προκύψει όταν το πρόγραμμα είναι σε εκμετάλλευση, τα πράγματα δεν είναι τόσο εύκολα.

19.6 "class" ή "struct";

Όπως είδαμε πιο πριν, τίποτε δεν μας εμποδίζει να ορίσουμε οποιαδήποτε κλάση είτε με `struct` είτε με `class`. Θα πρέπει όμως να έχουμε κάποιον τρόπο για να αποφασίζουμε κατά περίπτωση πώς θα κάνουμε τον ορισμό μας.

Τι διαφέρει το "`class`" από το "`struct`"; Όπως είπαμε:

- Όταν δίνουμε ορισμό κλάσης με το "`class`" τότε όλα τα μέλη είναι *εσωτερικά*, υπάρχει δηλαδή αυτόματη δήλωση `private` για τα πάντα. Ό,τι θέλουμε να ανοίξουμε προς τα έξω θα πρέπει να δηλωθεί `public`.
- Όταν δίνουμε ορισμό κλάσης με το "`struct`" τότε όλα τα μέλη είναι *ανοικτά*, υπάρχει δηλαδή αυτόματη δήλωση `public` για τα πάντα. Ό,τι θέλουμε να "κρύψουμε" θα πρέπει να δηλωθεί `private`.

Θα μπορούσαμε λοιπόν να πούμε ότι η C++ μας δίνει ένα κριτήριο για να πάρουμε την αποφασή μας: Αν έχουμε να «κρύψουμε» κάτι δίνουμε ορισμό με "`class`" αλλιώς με "`struct`". Αυτή είναι περίπου η απάντηση στο πρόβλημά μας.[6]

- Η δήλωση με `struct` χρησιμοποιείται συνήθως για απλές δομές όπου δεν έχουμε να κρύψουμε κάτι και όλα είναι ανοικτά. Αυτή ήταν η χρήση της και στη C που δεν είχε κλάσεις.
- Όταν έχουμε περιοχές `private` και `public` χρησιμοποιούμε τη δήλωση με `class`.

---

[5] *Δεν* θα δώσουμε τέτοιο παράδειγμα.

[6] Να υπενθυμίσουμε ότι και η C# βλέπει τις δομές ως «*κλάσεις με όλα τα μέλη ανοικτά που δεν κληρονομούν ούτε κληρονομούνται.*»

Να το πούμε και χρησιμοποιώντας την αναλλοίωτη:[7]

- ***Για να χρησιμοποιήσεις `struct` θα πρέπει να έχεις οπωσδήποτε τετριμμένη αναλλοίωτη (`true`).***

Παράδειγμα 1 ✍

Για τη

```
struct complex
{
   double re;
   double im;
   complex( double rp = 0.0, double ip = 0.0 )
   {  re = rp;  im = ip;  };
}; // complex
```

τι αναλλοίωτη να γράψουμε;

$$(\texttt{-DBL_MAX} \le re \le \texttt{DBL_MAX}) \wedge (\texttt{-DBL_MAX} \le im \le \texttt{DBL_MAX})$$

Αυτό όμως ισχύει πάντοτε για κάθε τιμή τύπου `double` και όχι ειδικώς για τα μέλη της *complex*. Εδώ λοιπόν η αναλλοίωτη είναι `true`.

✍✍✍

Αυτή όμως είναι αναγκαία αλλά όχι και ικανή συνθήκη για να επιλέξουμε τη `struct`.

Παράδειγμα 2 ✍

Θέλουμε να υλοποιήσουμε έναν τύπο συνόλων που στοιχεία τους θα είναι κεφαλαία γράμματα του λατινικού αλφαβήτου. Για την υλοποίηση επιλέγουμε τη χρήση *ψηφιοχάρτη*. Δηλαδή ένα σύνολο θα παριστάνεται με τα πρώτα 26 δυαδικά ψηφία μιας τιμής τύπου `long int` και:

- το δυαδικό ψηφίο 0 έχει τιμή 1 αν και μόνον αν το `'A'` ανήκει στο σύνολο,
- το δυαδικό ψηφίο 1 έχει τιμή 1 αν και μόνον αν το `'B'` ανήκει στο σύνολο,

  . . .
- το δυαδικό ψηφίο 25 έχει τιμή 1 αν και μόνον αν το `'Z'` ανήκει στο σύνολο.

Θα γράψουμε μια

```
class SetOfUCL
{
public:
// . . .
private:
   long int bitmap;
}; // SetOfUCL
```

ή μια `struct`;

Εδώ η αναλλοίωτη είναι `true`. Αλλά αν επιλέξουμε `struct` έχουμε το εξής πρόβλημα: Δίνουμε τη δυνατότητα στο πρόγραμμα που τη χρησιμοποιεί να χειρίζεται την τιμή του (μοναδικού) μέλους ενώ η κλάση γράφεται για να του δώσει τη δυνατότητα να χειρίζεται σύνολα και τα μέλη τους. Πώς αποφεύγεται κάτι τέτοιο; Με το να την κάνουμε `class`, να βάλουμε το `long int bitmap` («μυστικό» της υλοποίησης) σε περιοχή `private` και *να μην γράψουμε μεθόδους getBitmap ούτε setBitmap*.

✍✍✍

Ας πούμε λοιπόν ότι: Αν μια κλάση

1. έχει αναλλοίωτη `true` (τα πάντα δεκτά) και
2. δεν υπάρχουν «μυστικά» της υλοποίησης, δηλαδή μέλη στα οποία δεν θέλουμε να δώσουμε άμεση πρόσβαση

---

[7] Στο (Lockheed-Martin 2005) ο *AV Rule* 66 λέει: "A class **should** be used to model an entity that maintains an invariant", δηλαδή: για μια οντότητα που διατηρεί μια αναλλοίωτη **πρέπει** να χρησιμοποιείται μοντέλο `class` (και όχι `struct`).

θα προτιμούμε να ομαδοποιούμε τα στοιχεία μας με μια δομή.

Πάντως, όταν ορίζουμε μια `struct` μπορεί να ορίζουμε δημιουργό (ή δημιουργούς), να ορίζουμε συναρτήσεις-μέλη και να επιφορτώνουμε τελεστές αν αυτό μας διευκολύνει στην ανάπτυξη της εφαρμογής.

Και ένα ακόμη ερώτημα σχετικό με τα παραπάνω. Ας πούμε ότι έχουμε μια κλάση *Circle* που κάθε αντικείμενό της παριστάνει έναν κύκλο και έχει τρία μέλη: τις συντεταγμένες του κέντρου *cXC*, *cYC* και την ακτίνα *cR*. Φυσικά για την ακτίνα έχουμε $cR \geq 0$. Για το κέντρο όμως δεν έχουμε περιορισμό, θα μπορούσε να είναι οπουδήποτε. Τι κάνουμε σε αυτήν την περίπτωση; Βάζουμε `private` την ακτίνα και `public` τις συντεταγμένες του κέντρου; Όχι! Ο μεταγλωττιστής δεν θα έχει αντίρρηση για κάτι τέτοιο αλλά γενικώς θα τηρούμε τον κανόνα:[8]

♦ ***Σε μια κλάση που υλοποιείται με*** `class` ***τα μέλη είναι σε περιοχή*** `private`.

19.7 Από τη "`struct GrElmn`" στην "`class GrElmn`"

Ας εφαρμόσουμε τώρα αυτά που μάθαμε –και μερικά άλλα– στην κλάση *GrElmn* που είδαμε στην §15.14:[9]

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

Ποια είναι η αναλλοίωτη της κλάσης; Ας ξεκινήσουμε με την:

$$0 < geANumber < geAWeight$$

και να παρατηρήσουμε τα εξής:

- Αν δηλώσουμε

  ```
  GrElmn a;
  ```

  τι τιμές θα έχουν τα *a*.*geANumber*, *a*.*geAWeight*; Μια λογική επιλογή είναι: *a*.*geANumber* == *a*.*geAWeight* == 0.
- Το μήκος του *geSymbol* είναι 1 (π.χ. "U") ή 2 (π.χ. "Ca"). Αυτό θα πρέπει να περιληφθεί στην αναλλοίωτη.
- Τα *geSymbol* και *geName* έχουν μονον λατινικά γράμματα ενώ το *geGrName* μόνον ελληνικά. Και αυτό θα πρέπει να φαίνεται στην αναλλοίωτη.
- Στον ατομικό αριθμό δεν θα πρέπει να βάλουμε κάποιο απόλυτο (και χαμηλότερο) άνω όριο; Κάτι σαν 105 ή 112 ας πούμε; Ε, τώρα τέτοιο όριο δεν υπάρχει, διότι κάθε τόσο οι πυρηνικοί φυσικοί συνθέτουν στο εργαστήριο και έναν νέο (ασταθή) πυρήνα. Για το πρόγραμμά μας, το μόνο όριο που υπάρχει είναι αυτό που βάζει το περιεχόμενο του αρχείου μας.

  Γράφουμε λοιπόν την εξής αναλλοίωτη:

---

[8] Σχετική σύσταση του (CERT 2009) η OBJ00: "Declare data members `private`". Στο (Lockheed-Martin 2005) ο *AV Rule* 67 λέει: "`public` and `protected` data should only be used in `struct`s—not `class`es" (το "`protected`" θα το μάθουμε αργότερα).

[9] Πρόσεξε ότι την είχαμε γράψει εξ αρχής τηρώντας την ονοματολογική σύμβαση που δώσαμε παραπάνω

$$((0 < geANumber < geAWeight) \;||\; (geANumber == 0 \;\&\&\; geAWeight \geq 0) \;||\; (geAWeight == 0 \;\&\&\; geANumber \geq 0)) \;\&\&\; (geSymbol \leq 2)$$

Δηλαδή: αν τα *geANumber*, *geAWeight* έχουν αμφότερα μη μηδενικές τιμές θα πρέπει να ισχύει η 0 < *geANumber* < *geAWeight*· αν το ένα από τα δύο έχει τιμή 0 το άλλο θα πρέπει να έχει μη αρνητική τιμή. Οι μηδενικές τιμές θα επιτρέπονται μόνον από τον (ερήμην) δημιουργό αλλά όχι από τις σχετικές *set*.

Αφήνουμε ως άσκηση τους ελέγχους για λατινικά στα *geSymbol* και *geName* και ελληνικά στο *geGrName*.

Αφού η αναλλοίωτη δεν είναι τετριμμένη θα πρέπει να έχουμε υλοποίηση:

```
class GrElmn
{// I: ((0 < geANumber < geAWeight) ||
 //     (geANumber == 0 && geAWeight ≥ 0) ||
 //     (geAWeight == 0 && geANumber ≥ 0))
 // && (geSymbol ≤ 2)
public:
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
// . . .
private:
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[4];
   char               geName[14];
   char               geGrName[14];
}; // GrElmn
```

Ας ξεκινήσουμε με έναν («2 σε 1») δημιουργό: Δηλώνεται ως

```
   GrElmn( int aan=0, float aaw=0,
           string as="", string anm="", string agn="" );
```

και ορίζεται

```
GrElmn::GrElmn( int aan, float aaw,
                string as, string anm, string agn )
{
   if ( aan < 0 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::negANumber, aan );
   if ( aaw < 0 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::negANumber, aaw );
// aan >= 0 && aaw >= 0
   if ( (aan != 0 && aaw != 0) && aan >= aaw )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::an_gt_aw,
                        aan, aaw );
// (aan == 0 && aaw >=  0) || (aaw == 0 && aan >= 0) ||
// (0 < aan < aaw)
   geANumber = aan;
   geAWeight = aaw;
// (geANumber == 0 && geAWeight >= 0) ||
// (geAWeight == 0 && geANumber >= 0) ||
// (0 < geANumber < geAWeight)
   if ( as.length() > 2 )
      throw GrElmnXptn( "GrElmn", GrElmnXptn::longSymbol,
                        as.c_str() );
   strncpy( geSymbol, as.c_str(), symbolSz-1 );
   geSymbol[symbolSz-1] = '\0';
   strncpy( geName, anm.c_str(), nameSz-1 );
   geName[nameSz-1] = '\0';
   strncpy( geGrName, agn.c_str(), grNameSz-1 );
   geGrName[grNameSz-1] = '\0';
} // GrElmn::GrElmn
```

Αλλά, όπως είπαμε, στις *setAWeight()* και *setANumber()* δεν επιτρέπουμε τιμή 0:

```
void GrElmn::setAWeight( float aaw )
```

```
{
   if ( aaw <= 0 )
      throw GrElmnXptn( "setAWeight",
                        GrElmnXptn::negAWeight, aaw );
// aaw > 0
   if ( geANumber >= aaw )
      throw GrElmnXptn( "setAWeight", GrElmnXptn::an_gt_aw,
                        geANumber, aaw );
// 0 <= geANumber < aaw
   geAWeight = aaw;
// 0 <= geANumber < geAWeight
} // GrElmn::setAWeight
```

Δηλαδή: δεν επιτρέπουμε –με τη *setAWeight()*– να βάλουμε στο *geAWeight* τιμή 0. Το ίδιο πρέπει να κάνουμε και στην *setANumber()*:

```
void GrElmn::setANumber( int aan )
{
   if ( aan <= 0 )
      throw GrElmnXptn( "setANumber",
                        GrElmnXptn::negANumber, aan );
// aan > 0
   if ( geAWeight != 0 && geAWeight <= aan )
      throw GrElmnXptn( "setANumber",
                        GrElmnXptn::an_gt_aw, aan, geAWeight );
// (aan > 0) && (geAWeight != 0 ⇒ 0 < aan < geAWeight)
   geANumber = aan;
// (geANumber > 0) &&
// (geAWeight != 0 ⇒ 0 < geANumber < geAWeight)
} // GrElmn::setANumber
```

Παρατήρηση: ▶

Όπως θα δεις στη συνέχεια, για τα δύο προγράμματα της §15.14 που θα μετατρέψουμε, δεν θα χρειαστούμε μεθόδους *"set"*, εκτός από την *setGrName*. Ας σκεφθούμε όμως πόσο καλό είναι γενικώς το να έχουμε μια *setANumber()*.

Κατ' αρχάς να παρατηρήσουμε ότι δεν είναι δυνατόν να έχουμε δύο στοιχεία που να έχουν ίδιο *geANumber* ή *geSymbol* ή *geName* ή *geGrName*. Όλα αυτά τα μέλη είναι **υποψήφια κλειδιά** (candidate keys) για οποιαδήποτε συλλογή αντικειμένων κλάσης *GrElmn*. Ως κλειδί μας συμφέρει να επιλέξουμε το *geANumber* αφού είναι το πιο απλό. Οι μέθοδοι *"set"* για τροποποίηση των τιμών των *geSymbol*, *geName* και *geGrName* είναι μάλλον απαραίτητες αφού οι τιμές είναι κείμενα και υπάρχουν πολλές πιθανότητες για λάθη –ακόμη και ορθογραφικά– που θα πρέπει να διορθωθούν.

Η *setANumber()* όμως μπορεί να είναι ακόμη και επικίνδυνη:

- Αφού το *geANumber* είναι κλειδί, σε οποιαδήποτε συλλογή αντικειμένων *GrElmn* –όπως το περιεχόμενο του `elementsGr.dta`– υπάρχει το πολύ ένα αντικείμενο με κάποιο συγκεκριμένο κλειδί. Άρα:
  - Όταν εισάγουμε ένα αντικείμενο στη συλλογή θα πρέπει να ελέγχουμε αν υπάρχει άλλο αντικείμενο με το κλειδί του εισαγόμενου.
  - Αν γράψουμε μια *setANumber()* αυτή θα πρέπει να απενεργοποιείται όταν το αντικείμενο εισάγεται στη συλλογή.
- Αλλαγή της τιμής του *geANumber* σημαίνει αλλαγή στοιχείου· θα πρέπει να ακολουθείται από αλλαγές όλων των άλλων μελών. Είναι απλούστερο και ασφαλέστερο να χρησιμοποιείς τον δημιουργό. Αν, ας πούμε έχεις δώσει:

```
   GrElmn oneElmn( 20, 40.08, "Ca", "Calcium", "Ασβέστιο" );
```

και στη συνέχεια θέλεις να αποθηκεύσεις τα δεδομένα για τον άνθρακα δώσε:

```
   oneElmn = GrElmn( 6, 12.011, "C", "Carbon", "Άνθρακας" );
```

Αν κάνεις τις αλλαγές με τις *"set"* όλο και κάτι μπορεί να ξεφύγει.◀

Το πρώτο πρόγραμμα –της §15.14.1– χρησιμοποιεί μόνον δύο συναρτήσεις, τις

- *GrElmn_copyFromElmn()* και
- *GrElmn_save()*.

Βλέποντας κανείς τις

```
   fv.geANumber = a.eANumber;
   fv.geAWeight = a.eAWeight;
   strcpy( fv.geSymbol, a.eSymbol );
   strcpy( fv.geName, a.eName );
   fv.geGrName[0] = '\0';
```

θα σκεφθεί ότι για τη μετατροπή του προγράμματός μας στα νέα δεδομένα χρειαζόμαστε 5 μεθόδους "*set*".

Αλλά οι "*set*" γράφονται για να κάνουν έλεγχο στις τιμές που περνούν στα μέλη του αντικειμένου. Στην περίπτωσή μας, που παίρνουμε τα δεδομένα μας από ένα αρχείο που είναι ελεγμένο (είναι άλλωστε και μη μορφοποιημένο) αυτοί οι έλεγχοι είναι χάσιμο χρόνου. Θα δώσουμε λοιπόν μια άλλη λύση, πιο καλή, με έναν δεύτερο δημιουργό για την κλάση μας:

```
GrElmn::GrElmn( const Elmn& rhs )
{
   geANumber = rhs.eANumber;
   geAWeight = rhs.eAWeight;
   strcpy( geSymbol, rhs.eSymbol );
   strcpy( geName, rhs.eName );
   geGrName[0] = '\0';
} // GrElmn::GrElmn
```

Αυτός ο δημιουργός τροφοδοτείται με ένα αντικείμενο κλάσης *Elmn* και δημιουργεί ένα αντικείμενο κλάσης *GrElmn*· είναι ένας *δημιουργός μετατροπής*, όπως θα μάθουμε αργότερα. Έτσι, στο πρόγραμμά μας θα έχουμε:

```
   // Διάβασε μια εγγραφή
      Elmn oneElmn;
      int k( 0 );
      Elmn_load( oneElmn, bin );
      while ( !bin.eof() )
      {
      // Αντίγραψε την εγγραφή
         GrElmn oneGrElmn( oneElmn );
      // Γράψε τη νέα εγγραφή
         oneGrElmn.save( bout );
         ++k;
      // Διάβασε την επόμενη εγγραφή
         Elmn_load( oneElmn, bin );
      } // while
```

Η *GrElmn_save* θα μας δώσει την:

```
void GrElmn::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw GrElmnXptn( "save", GrElmnXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&geANumber),
               sizeof(geANumber) );
   bout.write( reinterpret_cast<const char*>(&geAWeight),
               sizeof(geAWeight) );
   bout.write( geSymbol, symbolSz );
   bout.write( geName, nameSz );
   bout.write( geGrName, grNameSz );
   if ( bout.fail() )
      throw GrElmnXptn( "save", GrElmnXptn::cannotWrite );
} // GrElmn::save
```

Για το δεύτερο πρόγραμμα –της §15.14.2– πέρα από τη *GrElmn::save()*, θα πρέπει να μετατρέψουμε σε μεθόδους και τις *GrElmn_load()*, *GrElmn_display()*, *GrElmn_setGrName()*. Η μετατροπή δεν έχει δυσκολίες:

```
void GrElmn::load( istream& bin )
```

```
{
   bin.read( reinterpret_cast<char*>(&geANumber),
             sizeof(geANumber) );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&geAWeight),
                sizeof(geAWeight) );
      bin.read( geSymbol, symbolSz );
      bin.read( geName, nameSz );
      bin.read( geGrName, grNameSz );
      if ( bin.fail() )
         throw GrElmnXptn( "load", GrElmnXptn::cannotRead );
   }
} // GrElmn::load

void GrElmn::display( ostream& tout ) const
{
   tout << "atomic number: " << geANumber << endl
        << "atomic weight: " << geAWeight << endl
        << "symbol: " << geSymbol << endl
        << "name: " << geName << endl
        << "greek name: " << geGrName << endl;
} // GrElmn::display

void GrElmn::setGrName( string newGrName )
{
   strncpy( geGrName, newGrName.c_str(), grNameSz-1 );
   geGrName[grNameSz-1] = '\0';
} // GrElmn::setGrName
```

Στη *writeRandom()* υπάρχει η εντολή:

```
bout.seekp( (a.geANumber-1)*GrElmn::saveSize );
```

Φυσικά, η συνάρτηση αυτή δεν έχει δικαίωμα πρόσβασης στο μέλος *geANumber* και θα χρειαστούμε τη σχετική "*get*":

```
unsigned short int getANumber() const {  return geANumber;  }
```

Έτσι, στη *writeRandom* θα έχουμε:

```
bout.seekp( (a.getANumber()-1)*GrElmn::saveSize );
```

Αν μετατρέψουμε σε μέθοδο και την *GrElmn_writeToTable()*, που είδαμε στην §15.14.2, παίρνουμε:

```
void GrElmn::writeToTable( ostream& tout ) const
{
   tout << geANumber << '\t' << geGrName << " (" << geName
        << ")\t" << geSymbol << '\t' << geAWeight << endl;
} // GrElmn::writeToTable
```

Τελικώς θα έχουμε στο `GrElmn.h`:

```
#ifndef _GRELMN_H
#define _GRELMN_H

#include <fstream>
#include <string>

using namespace std;

struct Elmn
{
   unsigned short int eANumber;     // ατομικός αριθμός
   float              eAWeight;     // ατομικό βάρος
   char               eSymbol[4];
   char               eName[14];
}; // Elmn

class GrElmn
{
```

```
public:
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   GrElmn( int aan=0, float aaw=0,
           string as="", string anm="", string agn="" );
   GrElmn( const Elmn& rhs );
   unsigned short int getANumber() const {  return geANumber;  }
   void setAWeight( float aaw );
   void setGrName( string newGrName );
   void save( ostream& bout ) const;
   void load( istream& bin );
   void display( ostream& tout ) const;
   void writeToTable( ostream& tout ) const;
private:
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn

struct GrElmnXptn
{
   enum { negANumber, negAWeight, an_gt_aw, longSymbol,
          fileNotOpen, cannotWrite, cannotRead };
   char  funcName[100];
   int   errorCode;
   float errFltVal1;
   float errFltVal2;
   char  errStrVal[100];
   GrElmnXptn( const char* mn, int ec, float ev1, int ev2=0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errFltVal1 = ev1;  errFltVal2 = ev2;  }
   GrElmnXptn( const char* mn, int ec, const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // GrElmnXptn

#endif // _GRELMN_H
```

και στο `GrElmn.cpp`:

```
#ifndef _GRELMN_CPP
#define _GRELMN_CPP

#include <fstream>
#include "GrElmn.h"

//========================= δημιουργοί =========================
GrElmn::GrElmn( int aan, float aaw,
                string as, string anm, string agn )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
GrElmn::GrElmn( const Elmn& rhs )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
//========================= setters ==========================
void GrElmn::setAWeight( float aaw )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::setGrName( string newGrName )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
//======================= άλλες μέθοδοι =======================
void GrElmn::save( ostream& bout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::load( istream& bin )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::display( ostream& tout ) const
```

```
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void GrElmn::writeToTable( ostream& tout ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _GRELMN_CPP
```

Η κλάση αυτή έχει όλα όσα χρειάζονται για να (ξανα)γράψουμε τα προγράμματα της §15.14. Δεν χρειαζόμαστε "*get*" και "*set*"; Χρειαζόμαστε! Μια κλάση γράφεται με αφορμή μια συγκεκριμένη εφαρμογή, αλλά συνήθως την εξοπλίζουμε ώστε να μπορεί να χρησιμοποιηθεί και αλλού. Στο επόμενο κεφάλαιο θα δούμε μια σχετική «συνταγή».

19.8 Μια Κλάση για Μπαταρίες[10]

Με δύο παραδείγματα μετάβασης από απλή δομή σε κλάση είδαμε, έστω και ακροθιγώς, μερικά βασικά πράγματα για τον αντικειμενοστρεφή προγραμματισμό. Τώρα θα δούμε άλλο ένα παράδειγμα όπου γράφουμε μια απλή κλάση από την αρχή (χωρίς να μετατρέψουμε κάτι που προϋπάρχει).

Το πρόβλημα:

Γράψε μια κλάση για την παράσταση μιας μπαταρίας. Ένα αντικείμενο-μπαταρία ξέρει την τάση του (σε volts), πόση ενέργεια (μέγιστη) μπορεί να αποθηκεύσει (σε joules) και πόσο είναι το τρέχον απόθεμα ενέργειας που έχει αποθηκευμένη (σε joules).

Πέρα από όποιες άλλες μεθόδους που θεωρείς απαραίτητες, να περιλάβεις, οπωσδήποτε και τις παρακάτω:

α) powerDevice(): Θα τροφοδοτείται με τον χρόνο t (σε sec), που θέλουμε να τροφοδοτεί μια συσκευή και το ρεύμα i (σε amperes) που τραβάει, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t). Θα επιστρέφει τιμή:

- `true` *αν η ενέργεια (τρέχουσα τιμή) που έχει η μπαταρία είναι αρκετή για να τροφοδοτήσει τη συσκευή. Στην περίπτωση αυτή θα αφαιρεί από την τρέχουσα τιμή της ενέργειας την ενέργεια που τράβηξε η συσκευή.*
- `false` *αν η ενέργεια δεν είναι αρκετή.*

β) maxTime(): Θα τροφοδοτείται με το ρεύμα i (σε amperes) που τραβάει μια συσκευή, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t) και θα επιστρέφει τον χρόνο (sec) που μπορεί να τροφοδοτεί τη συσκευή. Δεν αλλάζει το απόθεμα ενέργειας της μπαταρίας.

γ) recharge(): Θα επαναφορτίζει τη μπαταρία, δηλαδή θα βάζει το ενεργειακό απόθεμα στη μέγιστη τιμή του.

Γράψε πρόγραμμα που θα δοκιμάζει την κλάση που έγραψες: Θα δημιουργεί μπαταρία 12 *volts με μέγιστη δυνατότητα αποθήκευσης* 5×10^6 *joules. Στην αρχή θα τροφοδοτεί ένα λαμπτήρα που τραβάει* 4 *amperes για* 15 *min. Στη συνέχεια θα ερωτάται για τον χρόνο που μπορεί να τροφοδοτήσει μια συσκευή που τραβάει* 8 *amperes. Η ερώτηση θα υποβάλλεται ξανά αφού προηγουμένως η μπαταρία επαναφορτισθεί.*

Η αναλλοίωτη της κλάσης δεν μπορεί να είναι άλλη από την:

$$0 < bVoltage \;\&\&\; 0 < bMaxEnergy \;\&\&\; 0 \le bEnergy \le bMaxEnergy$$

Άρα ο ορισμός της κλάσης μας θα είναι περίπου ως εξής:

```
class Battery
{// I: 0 < bVoltage && 0 < bMaxEnergy && 0 ≤ bEnergy ≤ bMaxEnergy
public:
// ...
private:
   double bVoltage;   // volts
   double bMaxEnergy; // joules
   double bEnergy;    // joules
}; // Battery
```

[10] Από μια άσκηση του (Mansfield & Antonakos 1997).

Όπως βλέπεις, στα ονόματα τηρούμε τον κανόνα που είπαμε στην §19.1.4.

Η αντίστοιχη κλάση εξαιρέσεων είναι:

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

Για να αποφασίσουμε τι τιμές θα βάζει ο ερήμην δημιουργός μελετούμε το πρόβλημά μας. Εδώ βέβαια τα πράγματα είναι πολύ απλά: αφού μας ζητείται να δηλώσουμε μια 12βολτη μπαταρία με μέγιστη ενέργεια 5×10^6 *joules*, θα επωφεληθούμε και θα γράψουμε («2 σε 1») και τον δημιουργό με αρχικές τιμές. Θα δηλώσουμε μέσα στον ορισμό της κλάσης

```
   Battery( double v = 12, double me = 5e6 );
```

και θα ορίσουμε:

```
Battery::Battery( double v, double me )
{
   if ( v <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::voltageErr, v );
   if ( me <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::energyErr, me );
   bVoltage = v;
   bMaxEnergy = me;
   bEnergy = bMaxEnergy;
} // Battery::Battery
```

Όπως βλέπεις, αν παραβιασθεί η αναλλοίωτη από τις τιμές που καθορίζονται θα πρέπει να ρίχνονται οι κατάλληλες εξαιρέσεις. Πρέπει βέβαια να εισαχθούν στον `enum` της *BatteryXptn* τα *voltageErr* και *energyErr*.

19.8.1 Μέθοδοι "*get*", "*set*"

Χρειαζόμαστε μεθόδους "*get*" για να βλέπουμε τις τιμές μελών; Ναι, διότι θα γράψουμε μεν τη *maxTime()* (που έχει σαφώς περισσότερο νόημα από μια *getEnergy()* που πιθανότατα θα γράφαμε όπως κάναμε στα προηγούμενα παραδείγματα), αλλά θα χρειαστούμε μεθόδους για τα άλλα μέλη. Πού θα μας χρειαστούν; Ας πούμε ότι σε μια συνάρτηση έρχεται μια παράμετρος κλάσης *Battery*· πώς θα μάθουμε τα σταθερά χαρακτηριστικά της μπαταρίας; Γράφουμε λοιπόν τις:

```
   double getVoltage()  const {  return bVoltage;  };
   double getMaxEnergy()  const {  return bMaxEnergy;  };
```

Μεθόδους "*set*" χρειαζόμαστε; Θα πρέπει να κάνουμε σαφές ότι δεν θα πρέπει να υπάρχουν μέθοδοι που να αλλάζουν τις τιμές των *bVoltage* και *bMaxEnergy*, αφού αυτά παριστάνουν *σταθερά χαρακτηριστικά* της μπαταρίας που της αποδίδονται με την κατασκευή (δημιουργία) της. Η μόνη που έχει φυσικό νόημα είναι η *recharge()*, που θα γράψουμε στη συνέχεια.

19.8.2 Μέθοδος *powerDevice()*

«Θα τροφοδοτείται με τον χρόνο t (σε sec), που θέλουμε να τροφοδοτεί μια συσκευή και το ρεύμα i (σε amperes) που τραβάει, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t). Θα επιστρέφει τιμή:

- `true` *αν η ενέργεια (τρέχουσα τιμή) που έχει η μπαταρία είναι αρκετή για να τροφοδοτήσει τη συσκευή. Στην περίπτωση αυτή θα αφαιρεί από την τρέχουσα τιμή της ενέργειας την ενέργεια που τράβηξε η συσκευή.*
- `false` *αν η ενέργεια δεν είναι αρκετή.»*

Η *powerDevice()* θα αλλάζει την τιμή του αντικειμένου και θα επιστρέφει και μια τιμή τύπου `bool`. Δηλαδή, αν ήταν εξωτερική συνάρτηση θα ήταν κάπως έτσι:

```
??? Battery_powerDevice( Battery& ab,
                         double t, double i, bool& ok )
```

Αυτή έχει μια μεταβαλλόμενη παράμετρο (*ab*) και μια εξερχόμενη (*ok*). Οι κανόνες μας (§13.9) λένε ότι θα πρέπει να είναι `void`. Η μέθοδος θα είναι:

```
void Battery::powerDevice( double t, double i, bool& ok )
```

Οι προδιαγραφές της:

Προϋπόθεση: $t \geq 0$ && $i \geq 0$

Απαίτηση: (ok == ($bVoltage*i*t$ <= $bEnergy_{\text{αρχική}}$)) &&

($(ok \Rightarrow (bEnergy_{\text{τελική}}$ == $bEnergy_{\text{αρχική}}$ - $bVoltage*i*t))$)

Αν παραβιάζεται η προϋπόθεση, δηλαδή αν $t < 0$ ή $i < 0$ τότε ρίχνουμε εξαίρεση. *Αν η* $bVoltage*i*t$ <= $bEnergy_{\text{αρχική}}$ *δεν ισχύει δεν ρίχνουμε εξαίρεση.* Απλώς το πρόγραμμα-πελάτης πριν προχωρήσει σε οποιαδήποτε χρήση του αντικειμένου θα πρέπει να ελέγξει την τιμή της *ok*.

```
void Battery::powerDevice( double t, double i, bool& ok )
{
   if ( t < 0 )
      throw BatteryXptn( "powerDevice", BatteryXptn::timeErr, t );
   if ( i < 0 )
      throw BatteryXptn( "powerDevice",
                         BatteryXptn::currentErr, i );
   double reqEnergy( bVoltage*i*t );
   ok = reqEnergy <= bEnergy;
   if ( ok )  bEnergy -= reqEnergy;
} // Battery::powerDevice
```

Φυσικά, θα πρέπει να προσθέσουμε στην κλάση *BatteryXptn* τις δύο νέες σταθερές (*timeErr*, *currentErr*):

```
   enum { voltageErr, energyErr, timeErr, currentErr };
```

19.8.3 Μέθοδος *maxTime()*

«Θα τροφοδοτείται με το ρεύμα i (σε amperes) που τραβάει μια συσκευή, με τάση λειτουργίας, v, αυτήν της μπαταρίας (ενέργεια E = v·i·t) και θα επιστρέφει τον χρόνο (σε sec) που μπορεί να τροφοδοτεί τη συσκευή. Δεν αλλάζει το απόθεμα ενέργειας της μπαταρίας.»

Προδιαγραφές:

Προϋπόθεση: $i > 0$

Απαίτηση: $maxTime(i)$ == $bEnergy/(bVoltage*i)$

Θα μπορούσαμε να γράψουμε την εξής μέθοδο:

```
double Battery::maxTime( double i )  const
{
   if ( i <= 0 )
      throw BatteryXptn( "maxTime", BatteryXptn::currentErr, i );
   return bEnergy/(bVoltage*i);
} // Battery::maxTime
```

19.8.4 Μέθοδος *reCharge*

«Θα επαναφορτίζει τη μπαταρία, δηλαδή θα βάζει το ενεργειακό απόθεμα στη μέγιστη τιμή του.»

Προδιαγραφές:

Προϋπόθεση: `true`

Απαίτηση: *bEnergy* == *bMaxEnergy*

```
void Battery::reCharge()
{
   bEnergy = bMaxEnergy;
} // Battery::reCharge
```

19.8.5 Η Κλάση μας Τελικώς

Στο αρχείο `Battery.h` έχουμε:

```
#ifndef _BATTERY_H
#define _BATTERY_H

#include <string>

using namespace std;

class Battery
{// I: 0 < bVoltage && 0 < bMaxEnergy &&
 //    0 <= bEnergy <= bMaxEnergy
public:
   Battery( double v = 12, double me = 5e6 );
   double getVoltage()  const {  return bVoltage;  };
   double getMaxEnergy()  const {  return bMaxEnergy;  };
   void powerDevice( double t, double i, bool& ok );
   double maxTime( double i )  const;
   void reCharge();
private:
   double bVoltage;   // volts
   double bMaxEnergy; // joules
   double bEnergy;    // joules
}; // Battery

struct BatteryXptn
{
   enum { voltageErr, energyErr, timeErr, currentErr };
   char   funcName[100];
   int    errorCode;
   double errorValue;
   BatteryXptn( const char* mn, int ec, double ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errorValue = ev;  }
}; // BatteryXptn

#endif // _BATTERY_H
```

και στο `Battery.cpp`:

```
#ifndef _BATTERY_CPP
#define _BATTERY_CPP

#include <fstream>
#include "Battery.h"

Battery::Battery( double v, double me )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
void Battery::powerDevice( double t, double i, bool& ok )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
double Battery::maxTime( double i )  const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

```
void Battery::reCharge()
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

#endif // _BATTERY_CPP
```

19.8.6 Το Πρόγραμμα

Ας δούμε τώρα πώς θα είναι το πρόγραμμά μας:

«Γράψε πρόγραμμα που θα δοκιμάζει την κλάση που έγραψες: Θα δημιουργεί μπαταρία 12 volts με μέγιστη δυνατότητα αποθήκευσης $5 \cdot 10^6$ joules. Στην αρχή θα τροφοδοτεί έναν λαμπτήρα που τραβάει 4 amperes για 15 min. Στη συνέχεια θα ερωτάται για τον χρόνο που μπορεί να τροφοδοτήσει μια συσκευή που τραβάει 8 amperes. Η ερώτηση θα υποβάλλεται ξανά αφού προηγουμένως η μπαταρία επαναφορτισθεί.»

Μπορούμε να δημιουργήσουμε τη ζητούμενη μπαταρία είτε με την

```
   Battery btr;
```

είτε με την

```
   Battery btr( 12, 5e6 );
```

Στη συνέχεια τροφοδοτούμε τον λαμπτήρα με την (το πρώτο όρισμα θα πρέπει να είναι σε *sec*):

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

Παίρνουμε τον επιτρεπόμενο χρόνο τροφοδοσίας με την:

```
   cout << "μπορεί να τροφοδοτήσει με 8 A επί " << btr.maxTime(8)
        << " sec" << endl;
```

Τέλος, επαναφορτίζουμε και ξαναζητούμε τον χρόνο:

```
   btr.reCharge();
   cout << "μπορεί να τροφοδοτήσει με 8 A επί " << btr.maxTime(8)
        << " sec" << endl;
```

Φυσικά, θα πρέπει να βάλουμε και εντολές που συλλαμβάνουν και διαχειρίζονται τις εξαιρέσεις. Να ολόκληρο το πρόγραμμα:

```
#include <iostream>
#include <string>
#include "Battery.h"
#include "Battery.cpp"

using namespace std;

int main()
{
   bool ok;

   try
   {
      Battery btr( 12, 5e6 );

      btr.powerDevice( 15*60, 4, ok );
      cout << "μπορεί να τροφοδοτήσει με 8 A επί "
           << btr.maxTime(8) << " sec" << endl;

      btr.reCharge();
      cout << "μπορεί να τροφοδοτήσει με 8 A επί "
           << btr.maxTime(8) << " sec" << endl;
   }
   catch ( BatteryXptn& xpt )
   {
      switch ( xpt.errorCode )
      {
         case BatteryXptn::voltageErr:
            cout << xpt.funcName << ": Λάθος τάση ("
```

```
                << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::energyErr:
            cout << xpt.funcName << ": Λάθος ενέργεια ("
                << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::timeErr:
            cout << xpt.funcName << ": Λάθος χρόνος ("
                << xpt.errorValue << ')' << endl;    break;
         case BatteryXptn::currentErr:
            cout << xpt.funcName << ": Λάθος ρεύμα ("
                << xpt.errorValue << ')' << endl;    break;
         default:
              cout << xpt.funcName
                   << ": Μη αναμενόμενη εξαίρεση" << endl;
      } // switch
   } // catch
} // main
```

19.9 Τι (Πρέπει Να) Έμαθες στο Κεφάλαιο Αυτό

- Οι κλάσεις είναι τύποι δεδομένων που οι μεταβλητές τους (*αντικείμενα*) έχουν τη δυνατότητα να ελέγχουν το πώς τις διαχειρίζεται το περιβάλλον (πρόγραμμα) μέσα στο οποίο υπάρχουν και δρουν και να μην επιτρέπουν την οποιαδήποτε «κακομεταχείριση».
- Κάθε αντικείμενο έχει τα *μέλη* του και τις *συναρτήσεις-μέλη* ή *μεθόδους* του. Τα μέλη είναι συνήθως *κρυμμένα* από το περιβάλλον. Οι μέθοδοι έχουν πρόσβαση σε όλα τα μέλη του αντικειμένου.
- *Η διαχείριση κάθε αντικειμένου γίνεται μέσω μεθόδων* που έχει το κάθε αντικείμενο. *Οι μέθοδοι περιγράφουν και το πώς μπορεί να δράσει* το κάθε αντικείμενο. Οι μέθοδοι του κάθε αντικειμένου καθορίζονται όταν ορίζεται η κλάση στην οποίαν ανήκει.
- Οι τιμές των μελών κάθε αντικειμένου πληρούν μια συνθήκη που είναι η *αναλλοίωτη* της κλάσης.
- Στον ορισμό της κλάσης –και σε μορφή συνάρτησης που ονομάζεται *δημιουργός* ή *κατάσκευαστής*– της κλάσης μπορεί να περιλαμβάνονται οδηγίες για το πώς δημιουργείται ένα αντικείμενο.

Στη C++:

- Μια κλάση ορίζεται ως `struct` (τα πάντα ανοικτά) ή ως `class` (τα πάντα κρυμμένα). Και στις δύο περιπτώσεις μπορείς να καθορίζεις τι θα είναι τελικώς ανοικτό ή κρυμμένο καθορίζοντας περιοχές `public` και `private`.
- Ο ορισμός της κλάσης είναι ταυτοχρόνως και ορισμός ενός *ονοματοχώρου με το όνομα της κλάσης*.
- Τα μέλη δηλώνονται όπως οι μεταβλητές. Οι μέθοδοι δηλώνονται και ορίζονται όπως οι συναρτήσεις.
- Οι μέθοδοι ενός αντικειμένου έχουν πρόσβαση προς τα μέλη αντικειμένων της ίδιας κλάσης.

Αρχές καλού προγραμματισμού:

- Οι μέθοδοι κάθε κλάσης καθορίζονται από τις ανάγκες του *πελάτη* δηλαδή του προγράμματος που τη χρησιμοποιεί. (Αυτόν τον κανόνα θα τον αλλάξουμε κάπως στη συνέχεια.)
- Μαζί με κάθε κλάση ορίζουμε και αντίστοιχη *κλάση εξαιρέσεων*. Σε περίπτωση απόπειρας «κακομεταχείρισης» (παραβίασης της αναλλοίωτης) ενός αντικειμένου η «υπεύθυνη» μέθοδος ρίχνει εξαίρεση αυτής της κλάσης.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

19-1 Είδαμε ότι μπορούμε να γράψουμε με τρεις διαφορετικούς αλλά ισοδύναμους τρόπους:

```
struct Date
{
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

```
class Date
{
   unsigned int dYear;
// . . .
public:
   Date( . . . );
// . . .
}; // Date
```

```
class Date
{
public:
   Date( . . . );
// . . .
private:
   unsigned int dYear;
// . . .
}; // Date
```

Αλλά, με αυτά που είδαμε στη συνέχεια, μάλλον υπάρχουν και άλλοι ισοδύναμοι τρόποι.

1	2	3

`struct Date`
`{`
`public:`
`  Date( . . . );`
`// . . .`
`private:`
`  unsigned int dYear;`
`// . . .`
`}; // Date`
	`class Date`
	`{`
	`  Date( . . . );`
	`// . . .`
	`private:`
	`  unsigned int dYear;`
	`// . . .`
	`}; // Date`
		`struct Date`
		`{`
		`  unsigned int dYear;`
		`// . . .`
		`public:`
		`  Date( . . . );`
		`// . . .`
		`}; // Date`

Ισοδύναμοι με τους τρεις πρώτους είναι:

- Ο 1
- Ο 2
- Οι 1, 3
- Όλοι

19-2 Έστω ότι γράφουμε μια μέθοδο, *getDayOfWeek*, για τη *Date*, που επιστρέφει την ημέρα εβδομάδας για τη συγκεκριμένη ημερομηνία· δηλαδή μια τιμή τύπου:

```
typedef enum { sunday, monday, tuesday, wednesday, thursday,
               friday, saturday }  WeekDay;
```

Η δήλωσή της θα είναι:

- `WeekDay getDayOfWeek( Date d ) const;`
- `WeekDay getDayOfWeek( Date d );`
- `WeekDay getDayOfWeek() const;`
- `WeekDay getDayOfWeek();`
- `void getDayOfWeek( WeekDat& wd ) const;`

Β Ομάδα

19-3 Γράψε συνάρτηση *isValidDate* που θα τροφοδοτείται με τρεις ακέραιους *ay*, *am* και *ad* και θα επιστρέφει `true` αν αποτελούν έγκυρη ημερομηνία (αλλιώς `false`). Θα μπορούσαμε να την κάνουμε μέθοδο της της *Date*;
Υπόδ.: χρησιμοποίησε τον δημιουργό της *Date*.

19-4 Με «μπούσουλα» τη μετατροπή της *GrElmn*, μετάτρεψε σε κλάση τη δομή *Elmn*. Γράψε πρόγραμμα που θα δημιουργεί το αρχείο `elements.dta`. Προσπάθησε να βοηθήσεις τον χρήστη όσο πιο πολύ μπορείς ώστε να μην κάνει λάθη όταν κάνει εισαγωγή των δεδομένων. Ψάξε στο διαδίκτυο να βρεις πίνακα των χημικών στοιχείων στα αγγλικά για να κάνεις τα πειράματά σου.

Γ Ομάδα

19-5 *Με οδηγό τη λύση του προβλήματος της μπαταρίας λύσε το παρακάτω πρόβλημα:*

Θέλουμε μια κλάση:

```
class Piscina...
```

για να παριστάνουμε μια (θερμαινόμενη) πισίνα. Υποθέτουμε ότι η πισίνα είναι σχήματος ορθογωνίου παραλληλεπιπέδου και έχουμε μήκος (l), πλάτος (w) και βάθος (d) σε m. Οι πάγιες διαστάσεις είναι 6×4×2 αλλά μπορεί να είναι και άλλες (ό,τι θέλει ο πελάτης). Ένα χαρακτηριστικό της κατάστασης της πισίνας είναι το ύψος h (σε m) του νερού, που δεν μπορεί να είναι μεγαλύτερο από d – 0.3. Ο όγκος του νερού στην πισίνα σε m^3, είναι $l \cdot w \cdot h$. Ένα άλλο χαρακτηριστικό είναι η θερμοκρασία θ του νερού σε °C.

Η κλάση θα έχει ακόμη έναν ή περισσότερους δημιουργούς και τις εξής μεθόδσυς που θα πρέπει να γράψεις εσύ:

- *fill()*: Θα τροφοδοτείται με δύο πραγματικούς αριθμούς
  - *dh*, που θα παριστάνει την αύξηση του ύψους του νερού σε *m*,
  - *ta*, που θα παριστάνει τη θερμοκρασία (σε °C) του νερού που θα προστεθεί

  και θα επιστρέφει τον όγκο νερού που πρέπει να προσθέσουμε για να ανέβει το ύψος του νερού κατά *dh*. Θυμίσου ότι το ύψος του νερού δεν μπορεί να γίνει μεγαλύτερο από d – 0.3 *m*. Η θερμοκρασία του νερού μετά την πρόσθεση θα είναι: $(\theta * h + ta * dh) / (h + dh)$.
- *empty()*: Θα τροφοδοτείται με έναν πραγματικό αριθμό *dh* που θα παριστάνει την μείωση του ύψους του νερού σε *m* και θα επιστρέφει τον όγκο νερού που πρέπει να αδειάσουμε για να κατέβει το ύψος του νερού κατά *dh*. Φυσικά το ύψος του νερού δεν μπορεί να γίνει μικρότερο από 0.
- *heat()*: Θα τροφοδοτείται με έναν πραγματικό αριθμό *dt* που θα παριστάνει την επιθυμητή αύξηση της θερμικρασίας του νερού σε βαθμούς C και θα επιστρέφει την απαιτούμενη ενέργεια σε *cal*. Αυτή υπολογίζεται ως $V \cdot \rho \cdot c \cdot dt$ όπου V ο όγκος του νερού σε m^3, $\rho = 10^3\ kgr/m^3$ η πυκνότητα του νερού και $c = 10^3\ cal \cdot kgr^{-1} \cdot grad^{-1}$ η ειδική θερμότητα του νερού. Φυσικά, η θερμοκρασία του νερού δεν μπορεί να υπερβεί τους 100° C.

Γράψε πρόγραμμα όπου θα δηλώνονται δύο πισίνες: μια με τις πάγιες διαστάσεις και μια 5×3×1.2. Θα τις γεμίζει μέχρι τη μέση (1 και 0.6 *m* αντιστοίχως) με νερό θερμοκρασίας 10° C και θα θερμαίνει το νερό κατά 15° C. Μετά θα τις γεμίζει μέχρι το μέγιστο επιτρεπόμενο σημείο με νερό ίδιας θερμοκρασίας με αυτό που υπάρχει στην πισίνα. Τέλος, θα μας δίνει το νερό (σε m^3) και την ενέργεια (σε *cal*) που χρειάστηκε η κάθε μια.

project

3

Φοιτητές και Μαθήματα

Περιεχόμενα:

Prj03.1 Το Πρόβλημα 633
Prj03.2 Το (Πρώτο) Σχέδιο για το Πρόγραμμα 635
Prj03.3 Η Κλάση *Course* 636
Prj03.4 Η Κλάση *Student* 641
Prj03.5 Η Κλάση *StudentInCourse* 644
Prj03.6 Το Πρόγραμμα 645
Prj03.6.1 Η *loadCourses()* 646
Prj03.6.2 Η Ανάγνωση του Αρχείου των Δηλώσεων 647
Prj03.6.3 Τα Στοιχεία Ενός Φοιτητή 648
Prj03.6.4 . . . Και οι Δηλώσεις Μαθημάτων 649
Prj03.6.5 Η `main` 649
Prj03.7 `char*` ή *string*; 651
Prj03.8 Ένα Άλλο Σχέδιο για τις Κλάσεις 652
Ερωτήσεις - Ασκήσεις 653
Α Ομάδα 653

Prj03.1 Το Πρόβλημα

Μας δίνονται

- *ένα μη-μορφοποιημένο (binary) αρχείο με όνομα* **`gCourses.dta`** *με περιεχόμενο άγνωστο πλήθος αντικειμένων τύπου:*

```
class SylCourse
{
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
// . . .
private:
   char         cCode[cCodeSz];   // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   char         cPrereq[cCodeSz]; // προαπαιτούμενο
}; // SylCourse
```

που, όπως φαίνεται, περιγράφει ένα μάθημα προγράμματος σπουδών.

- *ένα μορφοποιημένο (text) αρχείο, με όνομα* **enrllmnt.txt**, *που περιέχει στοιχεία φοιτητών/τριών και δηλώσεων μαθημάτων στο τρέχον εξάμηνο. Ένα δείγμα περιεχομένου του αρχείου είναι:*

```
2149\tΜΠΡΟΥΜΟΥΤΗ\tΑΛΕΞΙΑ
4
ΕΥ0160Ε
ΕΥ0240Θ
ΕΥ0361Θ
ΤΠ0203Θ

2059\tΜΥΛΩΝΑΣ\tΣΤΑΥΡΟΣ
6
ΕΥ0221Θ
ΕΥ0240Ε
ΕΥ0331Ε
ΤΠ0101Θ
ΤΠ0305Ε
ΤΠ0314Θ
```

*Εδώ έχουμε στοιχεία και δηλώσεις μαθημάτων για δύο φοιτητές. Στην πρώτη γραμμή έχουμε τον αριθμό μητρώου του/της φοιτητή/τριας (π.χ. "*2149*" ή "*2059*"), μετά το επώνυμό του/της ("*ΜΠΡΟΥΜΟΥΤΗ*" ή "*ΜΥΛΩΝΑΣ*") και τέλος το όνομα ("*ΑΛΕΞΙΑ*" ή "*ΣΤΑΥΡΟΣ*"). Στην επόμενη γραμμή υπάρχει φυσικός αριθμός που είναι το πλήθος των μαθημάτων που δήλωσε ο/η φοιτητής/τρια. Στη συνέχεια ακολουθούν οι κωδικοί των μαθημάτων που δηλώθηκαν.*

Τα στοιχεία δύο φοιτητών διαχωρίζονται με μια κενή γραμμή.

Το αρχείο μαθημάτων είναι ελεγμένο. Το αρχείο δηλώσεων μπορεί να έχει λάθη δύο ειδών:

- *Περισσότερες από μια δηλώσεις του ίδιου φοιτητή.*
- *Λάθος κωδικό μαθήματος.*

Θέλουμε ένα πρόγραμμα που θα διαβάσει τα αρχεία που δίνονται και θα μας δώσει τέσσερα νέα:

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* **Students.dta**, *που θα έχει αντικείμενα τύπου*

```
class Student
{
public:
   enum { sNameSz = 20 };
// . . .
private:
   unsigned int sIdNum;            // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
   unsigned int sWH;               // ώρες ανά εβδομάδα
   unsigned int sNoOfCourses;      // αριθμός μαθημάτων που δήλωσε
}; // Student
```

ένα για κάθε φοιτητή.

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* **enrllmnt.dta**, *που θα έχει αντικείμενα τύπου*

```
class StudentInCourse
{
public:
// . . .
private:
   enum { sicCCodeSz = 8 };
   unsigned int sicSIdNum;            // αριθμός μητρώου
   char         sicCCode[sicCCodeSz]; // κωδικός μαθήματος
   float        sicMark;              // βαθμός στο μάθημα
}; // StudentInCourse
```

ένα για κάθε δήλωση (χωρίς λάθη) μαθήματος από φοιτητή.

- *Ένα μη-μορφοποιημένο αρχείο, με όνομα* `courses.dta`*, που θα έχει αντικείμενα τύπου Course, ένα για κάθε μάθημα. Η κλάση Course είναι σαν τη SylCourse με ένα επί πλέον μέλος:*

```
unsigned int cNoOfStudents;    // αριθ. φοιτητών στο μάθημα
```

  Φυσικά όλα τα αντικείμενα θα έχουν ενημερωμένο το νέο μέλος.

- *Ένα μορφοποιημένο αρχείο καταγραφής λαθών, με όνομα* `log.txt`*. Εκτός των λαθών που περιγράψαμε παραπάνω θα καταγράφονται και οι αριθμοί μητρώου των φοιτητών που δήλωσαν μαθήματα με άθροισμα ωρών μεγαλύτερο των 30 ανά εβδομάδα.*

Σημείωση: ▶

Το αρχείο `gCourses.dta` *έχει γραφεί με την παρακάτω μέθοδο:*

```
void SylCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw SylCourseXptn( "save", SylCourseXptn::fileNotOpen );
   bout.write( cCode, sizeof(cCode) );                    // κωδικός μαθήματος
   bout.write( cTitle, sizeof(cTitle) );                   // τίτλος μαθήματος
   bout.write( reinterpret_cast<const char*>(&cFSem), sizeof(cFSem) );
                                                           // τυπικό εξάμηνο
   bout.write( reinterpret_cast<const char*>(&cCompuls), sizeof(cCompuls) );
                                                       // υποχρεωτικό ή επιλογής
   bout.write( &cSector, sizeof(cSector) );                          // τομέας
   bout.write( cCateg, sizeof(cCateg) );                          // κατηγορία
   bout.write( reinterpret_cast<const char*>(&cWH), sizeof(cWH) );
                                                          // ώρες ανά εβδομάδα
   bout.write( reinterpret_cast<const char*>(&cUnits), sizeof(cUnits) );
                                                         // διδακτικές μονάδες
   bout.write( cPrereq, sizeof(cPrereq) );                  // προαπαιτούμενο
   if ( bout.fail() )
      throw SylCourseXptn( "save", SylCourseXptn::cannotWrite );
} // SylCourse::save
```

◀

Prj03.2 Το (Πρώτο) Σχέδιο για το Πρόγραμμα

Ας ξεκινήσουμε με δύο

Παρατηρήσεις ▶

1. Τα *cNoOfStudents* της *Course* και *sNoOfCourses*, *sWH* της *Student* φαίνονται κάπως «ξεκάφωτα». Και είναι! Όλα τα άλλα χαρακτηριστικά της *Course* είναι «πάγια» ενώ το *cNoOfStudents* αναφέρεται στο τρέχον εξάμηνο. Τα ίδια ισχύουν και για τα δύο χαρακτηριστικά της *Student*. Η σχεδίαση των κλάσεων δεν είναι σωστή! Παρ' όλα αυτά θα επιμείνουμε στη συγκεκριμένη περιγραφή για εκπαιδευτικούς λόγους: *Εδώ ενδιαφερόμαστε μόνο για την προγραμματιστική διαχείριση των κλάσεων.* Η σχεδίαση είναι αντικείμενο άλλων τομέων της τεχνολογίας λογισμικού.
2. Ένα αντικείμενο της *StudentInCourse* θα πρέπει να έχει όλες τις πληροφορίες που σχετίζονται με την εγγραφή ενός φοιτητή σε ένα μάθημα, π.χ.: δύο βαθμούς (για τις δύο εξεταστικές περιόδους) και όχι έναν, ακαδημαϊκό εξάμηνο (αν θέλουμε να έχουμε πολλά εξάμηνα) κλπ. Εδώ βάλαμε μόνον έναν βαθμό, για να απλουστεύσουμε τα πράγματα.◀

Η διαφορά αυτού του παραδείγματος από το «παράδειγμα της μπαταρίας» είναι ότι εκεί είχαμε μια κλάση και η διατύπωση του προβλήματος καθόριζε τις μεθόδους που έπρεπε να αναπτυχθούν. Εδώ, που δεν έχουμε τέτοια βοήθεια, πώς θα βρούμε τις μεθόδους που χρειαζόμαστε; Κάνουμε ένα σχέδιο του προγράμματος και από εκεί εντοπίζουμε τις ανάγκες μας. Υπάρχουν βέβαια ορισμένα σχεδόν απαραίτητα συστατικά για οποιαδήποτε κλάση που σχετίζονται με το πρόγραμμα στο οποίο θα χρησιμοποιηθεί. Θα τα μάθουμε στα επόμενα μαθήματα.

Ας κάνουμε λοιπόν ένα πρώτο σχέδιο για να δούμε τι γίνεται:

```
Φόρτωσε τον πίνακα των μαθημάτων
Μηδένισε το cNoOfStudents του κάθε μαθήματος
Άνοιξε το αρχείο enrllmnt.txt
do {
   Διάβασε τα στοιχεία ενός φοιτητή
   if ( δεν τελείωσε το αρχείο )
   {
      if ( υπάρχει στον πίνακα φοιτητών )
      {
         Αγνόησε τη δήλωση // υπάρχει ήδη άλλη δήλωση
         Γράψε στο log
      }
      else
      {
         Βάλε τον φοιτητή στον πίνακα φοιτητών
         Μηδένισε τα sNoOfCourses και sWH του φοιτητή
         for ( όλα τα μαθήματα που δήλωσε ο φοιτητής )
         {
            αναζήτησε τον κωδικό του μαθήματος
                                              στον πίνακα μαθημάτων
            if ( δεν υπάρχει )
               Γράψε στο log
            else
            {
               Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος)
                                               στον πίνακα δηλώσεων
               Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή
               Ενημέρωσε το cNoOfStudents του κάθε μαθήματος
            }
         } // for
      } // if ( υπάρχει στον πίνακα φοιτητών )
   } // if (δεν τελείωσε το αρχείο )
} while ( δεν τελείωσε το αρχείο )
for ( όλους τους φοιτητές )
   if ( sWH > 30 ) Γράψε στο log
Φύλαξε τα στοιχεία του πίνακα φοιτητών στο Students.dta
Φύλαξε τα στοιχεία του πίνακα δηλώσεων στο enrllmnt.dta
Φύλαξε τα στοιχεία του πίνακα μαθημάτων στο gCourses.dta
```

Στη συνέχεια θα καταγράψουμε τις απαιτήσεις που προκύπτουν για κάθε κλάση.

Prj03.3 Η Κλάση *Course*

Τι πρέπει να κάνουμε με την *Course*;

- Να διαβάζουμε στοιχεία αντικειμένων της κλάσης από μη-μορφοποιημένο αρχείο («*Φόρτωσε τον πίνακα των μαθημάτων*»). Αν υλοποιήσουμε την κλάση *SylCourse* θα πρέπει να γράψουμε
  - μια μέθοδο, ας την πούμε *load*, που θα φορτώνει τα στοιχεία ενός μαθήματος από (μη-μορφοποιημένο) αρχείο και
  - μια μέθοδο (ή, καλύτερα, επιφόρτωση του τελεστή εκχώρησης) που θα αντιγράφει ένα αντικείμενο *SylCourse* σε ένα αντικείμενο *Course*.

  Κάτι τέτοιο όμως είναι έξω από τους στόχους μας.[1] Αντί για αυτό θα γράψουμε μια συνάρτηση –ας την πούμε *loadSylCourse*– που θα διαβάζει τις τιμές μελών του αντικειμενου *SylCourse* και θα τις περνάει μια προς μία στα αντίστοιχα μέλη ενός αντικειμένου *Course*. Αυτό το «πέρασμα τιμών» θα πρέπει να γίνει με μεθόδους που θα δούμε στη συνέχεια.

[1] Αργότερα θα υλοποιήσουμε την κλάση *SylCourse* (με όνομα *Course*).

- Να μπορούμε να μηδενίζουμε τον μετρητή («*Μηδένισε το cNoOfStudents του κάθε μαθήματος*»). Αυτό μπορεί να γίνει είτε με μια μέθοδο *clearStudents()* που κάνει μόνον αυτό είτε με μια *setNoOfStudents()* που είναι γενικότερη.
- Να βρίσκουμε ένα αντικείμενο με συγκεκριμένο κωδικό («*Αναζήτησε τον κωδικό του μαθήματος στον πίνακα μαθημάτων*»). Επειδή η αναζήτηση θα γίνεται με βάση τον κωδικό θα χρειαστούμε μια μέθοδο που να μας δίνει τον κωδικό (*getCode()*).
- Να παίρνουμε τις ώρες διδασκαλίας ανά εβδομάδα για να τις προσθέσουμε στις αντίστοιχες του φοιτητή («*Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή*»). Θα πρέπει να γράψουμε μια *getWH*.
- Να αυξάνουμε τον αριθμό των φοιτητών που γράφονται στο μάθημα κατά 1 («*Ενημέρωσε το cNoOfStudents του κάθε μαθήματος*»). Μπορούμε είτε να γράψουμε μια *add1Student()* που αυξάνει την τιμή του *cNoOfStudents* κατά 1 είτε να γράψουμε μια *getNoOfStudents()* και να τη χρησιμοποιήσουμε μαζί με τη *setNoOfStudents()*.
- Να γράφουμε στοιχεία αντικειμένων της κλάσης σε μη-μορφοποιημένο αρχείο («*Φύλαξε τα στοιχεία του πίνακα μαθημάτων στο* `courses.dta`»). Ας πούμε *save* τη σχετική μέθοδο.

Ξεκινούμε με τη *loadSylCourse()*. Όπως μπορούμε να δούμε στη δήλωση της *SylCourse* έχουμε:

- Ένα μέλος τύπου `bool` (*cCompuls*).
- Τρία μέλη τύπου `unsigned int` (*cFSem*, *cWH*, *cUnits*).
- Πέντε μέλη τύπου `char` (*cCode* και *cPrereq* με μήκος *cCodeSz* (8), *cTitle* με μήκος *cTitleSz* (80), *cCateg* με μήκος *cCategSz* (4) και *cSector* με μήκος 1).

Αν λοιπόν δηλώσουμε τρεις ενταμιευτές:[2]

```
bool bBuf;
unsigned int iBuf;
char cBuf[Course::cTitleSz];
```

μπορούμε να διαβάσουμε αυτά που γράφηκαν με τη *SylCourse*::*save* με τις εξής εντολές:

```
bin.read( cBuf, Course::cCodeSz );                    // κωδικός μαθήματος
bin.read( cBuf, Course::cTitleSz );                    // τίτλος μαθήματος
bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                      // τυπικό εξάμηνο
bin.read( reinterpret_cast<char*>(&bBuf), sizeof(bool) );
                                                 // υποχρεωτικό ή επιλογής
bin.read( cBuf, sizeof(char) );                                // τομέας
bin.read( cBuf, Course::cCategSz );                          // κατηγορία
bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                      // ώρες ανά εβδομάδα
bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                     // διδακτικές μονάδες
bin.read( cBuf, Course::cCodeSz );                    // προαπαιτούμενο
```

Η *loadSylCourse()* θα πρέπει να είναι:

```
void loadSylCourse( ifstream& bin, Course& oneCourse )
{
   bool bBuf;
   unsigned int iBuf;
   char cBuf[Course::cTitleSz];

   bin.read( cBuf, Course::cCodeSz );                 // κωδικός μαθήματος
   if ( !bin.eof() )
   {
                                                 oneCourse.setCode( cBuf );
```

[2] και με την προϋπόθεση ότι και στην *Course* θα δηλώσουμε:

```
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
```

```
      bin.read( cBuf, Course::cTitleSz );          oneCourse.setTitle( cBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                    oneCourse.setFSem( iBuf );
      bin.read( reinterpret_cast<char*>(&bBuf), sizeof(bool) );
                                                  oneCourse.setCompuls( bBuf );
      bin.read( cBuf, sizeof(char) );           oneCourse.setSector( cBuf[0] );
      bin.read( cBuf, Course::cCategSz );          oneCourse.setCateg( cBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                      oneCourse.setWH( iBuf );
      bin.read( reinterpret_cast<char*>(&iBuf), sizeof(unsigned int) );
                                                   oneCourse.setUnits( iBuf );
      bin.read( cBuf, Course::cCodeSz );          oneCourse.setPrereq( iBuf );
      if ( bin.fail() )
         throw ProgXptn( "loadSylCourse", ProgXptn::cannotRead );
   }
} // loadSylCourse
```

Όπως βλέπεις, θα χρειαστούμε μεθόδους *setCode*() *setFSem(), setCompuls(), setSector(), setCateg(), setWH(), setUnits()* και *setPrereq()* για να αλλάζουμε (καθορίζουμε) τις τιμές των *cCode, cTitle, cFSem, cCompuls, cSector, cCateg, cWH, cUnits* και *cPrereq* αντιστοίχως.

Αυτές οι μέθοδοι θα πρέπει να γραφούν προσεκτικά διότι δεν θα πρέπει να αφήνουν «σκουπίδια» να περνούν μέσα στα αντικείμενα. Αργότερα θα δούμε πώς κάνουμε αυτήν τη δουλειά με πιο συστηματικό τρόπο. Εδώ, έχοντας τη σιγουριά ότι διαβάζουμε ένα αρχείο με σωστά στοιχεία, θα γράψουμε μεθόδους που θα περνούν τα στοιχεία χωρίς ελέγχους. Αλλά, για να δεις πώς περίπου γίνονται αυτοί οι έλεγχοι, θα γράψουμε

- Τη *setFSem()* έτσι ώστε να δέχεται ως τυπικό εξάμηνο διδασκαλίας του μαθήματος κάτι μεταξύ 1 και 8.

```
void Course::setFSem( int aFSem )
{
   if ( aFSem < 1 || 8 < aFSem)
      throw CourseXptn( "setFSem", CourseXptn::rangeError, aFSem );
   cFSem = aFSem;
} // Course::setFSem
```

- Τη *setPrereq()* έτσι ώστε να μην επιτρέπει ένα μάθημα να φέρεται ως προαπαιτούμενο του εαυτού του.

```
void Course::setPrereq( const string& prCode )
{
   if ( (prCode.length() > 0) && (prCode == cCode) )
      throw CourseXptn( "setPrereq", CourseXptn::autoRef, prCode.c_str() );
   strncpy( cPrereq, prCode.c_str(), cCodeSz-1 );  cPrereq[cCodeSz-1] = '\0';
} // Course::setPrereq
```

Φυσικά, για να διασφαλίσουμε ότι ένα μάθημα δεν φέρεται ως προαπαιτουμενο του εαυτού του θα πρέπει να προσέξουμε και τη

```
void Course::setCode( string aCode )
{
   if ( (aCode.length() > 0) && (aCode == cPrereq) )
      throw CourseXptn( "setCode", CourseXptn::autoRef, aCode.c_str() );
   strncpy( cCode, aCode.c_str(), cCodeSz-1 );        cCode[cCodeSz-1] = '\0';
} // Course::setCode
```

Οι υπόλοιπες, οι «απρόσεκτες» προς το παρόν, θα είναι:

```
void Course::setTitle( string aTitle )
{
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
} // Course::setTitle
void Course::setSector( char aSector )  {  cSector = aSector;  }
void Course::setCateg( string aCateg )
{
   strncpy( cCateg, aCateg.c_str(), cCategSz-1 );  cTitle[cCategSz-1] = '\0';
} // Course::setCateg
void Course::setWH( int aWH )  {  cWH = aWH;  }
```

```
void Course::setUnits( int aUnits )  {  cUnits = aUnits;  }
```

Ας δούμε τώρα τα υπόλοιπα· και αρχίζουμε από τα εύκολα:

```
   unsigned int getNoOfStudents() {  return cNoOfStudents;  }
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   unsigned int getWH() const {  return cWH;  }
```

Γιατί δεν γράψαμε μια *setNoOfStudents*; Διότι αυτές που γράψαμε είναι πιο λειτουργικές: Αρχικώς, για κάποιο μάθημα "`Course ac`", βάζουμε:

```
   ac.clearStudents();
```

και κάθε φορά που βρίσκουμε φοιτητή που το έχει δηλώσει δίνουμε:

```
   ac.add1Student();
```

Αν υλοποιούσαμε τη *setNoOfStudents* θα είχαμε αντιστοίχως:

```
   ac.setNoOfStudents( 0 );
```

και:

```
   ac.setNoOfStudents( ac.getNoOfStudents()+1 );
```

Αλλά, η *setNoOfStudents* χρειάζεται έλεγχο (για αρνητικές τιμές) ενώ και οι μη αρνητικές είναι «προβληματικές» (τι νόημα έχει `ac.setNoOfStudents(20);`) Η *getNoOfStudents()* χρειάζεται γενικότερα.

Εύκολη είναι και η

```
   const char* getCode() const {  return cCode;  }
```

αλλά εδώ χρειάζονται εξηγήσεις για το πρώτο "`const`". Γιατί χρειάζεται; Βγάλε τα δύο "`const`" και δοκίμασε τα παρακάτω:

```
   cout << ac.getCode() << endl;
   char* p( ac.getCode() );
   p[3] = 'X';  p[4] = 'Z';
   cout << ac.getCode() << endl;
```

Αποτέλεσμα:

```
EY0140E
EY0XZ0E
```

Δηλαδή, ανοίξαμε μια «κερκόπορτα» από όπου τροποποιείται αυθαιρέτως το αντικείμενό μας.

Κρατώντας τα δύο "`const`" (που πάνε «πακέτο») το μόνο που μπορείς να κάνεις με το *p* είναι:

```
   char* p( new char[strlen(ac.getCode())+1] );
   strcpy( p, ac.getCode() );
```

Τώρα ο *p* δείχνει *αντίγραφο* του *ac.cCode* και δεν έχει σχέση με το αντικείμενό μας.

Ας έλθουμε τώρα στις *load()* και *save()*. Η *save()* θα είναι σαν τη *SylCourse::save()*, που μας δόθηκε, αλλά θα φυλάγει και την τιμή του *cNoOfStudents*:

```
void Course::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw CourseXptn( "save", CourseXptn::fileNotOpen );
   bout.write( cCode, sizeof(cCode) );                    // κωδικός μαθήματος
   bout.write( cTitle, sizeof(cTitle) );                   // τίτλος μαθήματος
   bout.write( reinterpret_cast<const char*>(&cFSem), sizeof(cFSem) );
                                                            // τυπικό εξάμηνο
   bout.write( reinterpret_cast<const char*>(&cCompuls), sizeof(cCompuls) );
                                                       // υποχρεωτικό ή επιλογής
   bout.write( &cSector, sizeof(cSector) );                         // τομέας
   bout.write( cCateg, sizeof(cCateg) );                          // κατηγορία
   bout.write( reinterpret_cast<const char*>(&cWH), sizeof(cWH) );
                                                          // ώρες ανά εβδομάδα
   bout.write( reinterpret_cast<const char*>(&cUnits), sizeof(cUnits) );
                                                         // διδακτικές μονάδες
   bout.write( cPrereq, sizeof(cPrereq) );                  // προαπαιτούμενο
```

```
   bout.write( reinterpret_cast<const char*>(&cNoOfStudents),
               sizeof(cNoOfStudents) );              // αριθ. φοιτ. στο μάθημα
   if ( bout.fail() )
      throw CourseXptn( "save", CourseXptn::cannotWrite );
} // Course::save
```

Η *load()* θα γραφεί με οδηγό τη *save*, αφού θα διαβάσουμε όπως γράψαμε:

```
void Course::load( istream& bin )
{
   bin.read( cCode, cCodeSz );                           // κωδικός μαθήματος
   if ( !bin.eof() )
   {
      bin.read( cTitle, cTitleSz );                       // τίτλος μαθήματος
      bin.read( reinterpret_cast<char*>(&cFSem), sizeof(cFSem) );
                                                          // τυπικό εξάμηνο
      bin.read( reinterpret_cast<char*>(&cCompuls), sizeof(cCompuls) );
                                                        // υποχρεωτικό ή επιλογής
      bin.read( &cSector, sizeof(cSector) );                        // τομέας
      bin.read( cCateg, cCategSz );                              // κατηγορία
      bin.read( reinterpret_cast<char*>(&cWH), sizeof(cWH) );
                                                        // ώρες ανά εβδομάδα
      bin.read( reinterpret_cast<char*>(&cUnits), sizeof(cUnits) );
                                                      // διδακτικές μονάδες
      bin.read( cPrereq, cCodeSz );                        // προαπαιτούμενο
      bin.read( reinterpret_cast<char*>(&cNoOfStudents),
                sizeof(cNoOfStudents) );              // αριθ. φοιτ. στο μάθημα
      if ( bin.fail() )
         throw CourseXptn( "load", CourseXptn::cannotRead );
   } // if ( !bin.eof() ). . .
} // Course::load
```

Αυτή είναι η *load()* αλλά δεν είναι καλή. Γιατί; Όταν δίνεις την εντολή:

```
   ac.load( bin );
```

το αντικείμενο *ac* έχει κάποια τιμή. Αν κάτι «πάει στραβά» κατά την ανάγνωση αυτή η τιμή καταστρέφεται. Θα πρέπει να διαβάζουμε πιο προσεκτικά, όπως θα μάθουμε αργότερα. Προς το παρόν δεν θα έχουμε πρόβλημα, όπως θα δεις στη συνέχεια.

Για να μπορέσεις να κάνεις δοκιμές με την κλάση την εξοπλίζουμε και με μια:

```
void Course::display( ostream& tout ) const
{
   if ( tout.fail() )
      throw CourseXptn( "display", CourseXptn::fileNotOpen );
   tout << cCode << '\t' << cTitle << '\t';
   if ( cCompuls )  tout << "Y\t";
              else  tout << "YE\t";
   tout << cSector << '\t' << cCateg << '\t' << cWH << '\t'
        << cUnits << '\t' << cPrereq << '\t' << cFSem << '\t'
        << cNoOfStudents << endl;
   if ( tout.fail() )
      throw CourseXptn( "display", CourseXptn::cannotWrite );
} // Course::display
```

Να λοιπόν η «προχειρογραμμένη»

```
class Course
{
public:
   enum { cCodeSz = 8, cTitleSz = 80, cCategSz = 4 };
   Course( string aCode="" );
   const char* getCode() const {  return cCode;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) { cCompuls = aCompuls; };
   void setSector( char aSector )  {  cSector = aSector;  };
   void setCateg( string aCateg );
```

```
   void setWH( int aWH )  {  cWH = aWH;  };
   void setUnits( int aUnits )  {  cUnits = aUnits;  };
   void setPrereq( const string& prCode );
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   void save( ostream& bout ) const;
   void load( istream& bin );
   void display( ostream& tout ) const;
private:
   char         cCode[cCodeSz];   // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   char         cPrereq[cCodeSz]; // προαπαιτούμενο
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
}; // Course
```

και η αντίστοιχη κλάση εξαιρέσεων:

```
struct CourseXptn
{
   enum {  rangeError, autoRef,
           fileNotOpen, cannotWrite, cannotRead  };
   char funcName[100];
   int  errorCode;
   int  errIntVal;
   char errStrVal[100];
   CourseXptn( const char* mn, int ec, const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const char* mn, int ec, int iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errIntVal = iv;  }
}; // CourseXptn
```

Χρήσιμο είναι να ορίσουμε:

```
typedef Course* PCourse;
```

Prj03.4 Η Κλάση *Student*

Να δούμε τώρα τι απαιτήσεις έχουμε για τη *Student*.

- Η «*Διάβασε τα στοιχεία ενός φοιτητή*» παραπέμπει σε διάφορες εναλλακτικές λύσεις:
  - Να γράψουμε μια *readFromText* που διαβάζει τα στοιχεία από αρχείο –σαν τη *load*– αλλά, αυτήν τη φορά, μορφοποιημένο (text) ή
  - να γράψουμε τρεις "*set*": *setIdNum*, *setSurname*, *setFirstname* ή
  - να γράψουμε έναν δημιουργό

```
Student( int aIdNum=0, string aSurn="", string aFirstn="" )
```

- Για να κάνουμε τον έλεγχο «`if` (*υπάρχει στον πίνακα φοιτητών*)» πρέπει να αναζητούμε ένα αντικείμενο κλάσης *Student*. Επειδή η αναζήτηση θα γίνεται με βάση τον αριθμό μητρώου, θα χρειαστούμε μια μέθοδο που θα μας τον δίνει (*getIdNum*). Ανάγκη για την ίδια μέθοδο προκύπτει και από την «*Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος) στον πίνακα δηλώσεων*». Θα χρειαστούμε και εδώ έναν δημιουργό για αναζήτηση με τη *linSearch()*; Όχι, διότι στην περίπτωση αυτήν αναζητούμε ένα αντικείμενο που έχουμε διαβάσει από το αρχείο. Πάντως η επιφόρτωση του "`!=`" είναι αναγκαία.

- Ακολουθώντας το σκεπτικό που αναπτύξαμε για τη διαχείριση του μέλους *cNoOfStudents* της *Course*, από την «*Μηδένισε τα sNoOfCourses και sWH του φοιτητή*» συνάγουμε ανάγκη για μέθοδο *clearCourses()*.
- Για να ενημερώσουμε τους δύο μετρητές («*Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή*») θα χρειαστούμε μέθοδο *add1Course()*.
- Για να ελέγξουμε τις εβδομαδιαίες ώρες του φοιτητή («`if (sWH > 30)`») θα χρειαστούμε μια *getWH()*.
- Τέλος, θα χρειαστούμε μια μέθοδο, ας την πούμε *save()*, για να γράφουμε στοιχεία αντικειμένων της κλάσης σε μη-μορφοποιημένο αρχείο («*Φύλαξε τα στοιχεία του πίνακα φοιτητών στο* `Students.dta`»).

Ξεκινώντας και εδώ από τα εύκολα, έχουμε:

```
   unsigned int getIdNum() const {  return sIdNum;  }
   unsigned int getWH() const {  return sWH;  }
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
   void add1Course( int aIncrWH );
```

όπου:

```
void Student::add1Course( int aIncrWH )
{
   if ( aIncrWH <= 0 )
      throw StudentXptn( "add1Course", StudentXptn::nonPosIncr,
                         aIncrWH );
   ++sNoOfCourses;
   sWH += aIncrWH;
} // Student::add1Course
```

Ο αριθμός μητρώου είναι ένα άλλο παράδειγμα *υποκατάστατου κλειδιού*. Μπορούμε λοιπόν να επιφορτώσουμε τον "`!=`" με σύγκριση αριθμών μητρώου μόνο:

```
bool operator!=( const Student& a, const Student& b )
{  return ( a.getIdNum() != b.getIdNum() );  }
```

Για το διάβασμα θα επιλέξουμε –για εκπαιδευτικούς λόγους, για να δεις τη διαφορά από τη *load()*– να γράψουμε μια:

```
void Student::readFromText( istream& tin )
{
   string line;
   getline( tin, line, '\n' );
   if ( !tin.eof() )
   {
      size_t t1Pos( line.find("\t") );
      if ( t1Pos >= line.length() )
         throw StudentXptn( "readFromText",
                            StudentXptn::incomplete );
      size_t t2Pos( line.find("\t", t1Pos+1) );
      if ( t2Pos >= line.length() )
         throw StudentXptn( "readFromText",
                            StudentXptn::incomplete );
      string str1( line.substr(0, t1Pos) );
      int    iStr1;
      iStr1 = atoi( str1.c_str() );
      if ( iStr1 <= 0 )
         throw StudentXptn( "readFromText",
                            StudentXptn::negIdNum, iStr1 );
      sIdNum = iStr1;
      str1 = line.substr( t1Pos+1, t2Pos-t1Pos-1 );
      strncpy( sSurname, str1.c_str(), sNameSz-1 );
                                  sSurname[sNameSz-1] = '\0';
      str1 = line.substr( t2Pos+1 );
      strncpy( sFirstname, str1.c_str(), sNameSz-1 );
                                sFirstname[sNameSz-1] = '\0';
      sNoOfCourses = 0;
      sWH = 0;
```

```
   } // if ( !tin.eof . . .
} // Student::readFromText
```

Εδώ πρόσεξε τα εξής:

- Αν διαβάσουμε γραμμή που δεν έχει δύο '`\t`' ρίχνουμε εξαίρεση με κωδικό σφάλματος "*incomplete*". Γιατί το βάλαμε αυτό; Αφού έχουμε τη διαβεβαίωση ότι δεν υπάρχουν λάθη τέτοιου είδους. Η εμβέλεια της μεθόδου είναι πέρα από την τρέχουσα εφαρμογή.
- Ένα πρόβλημα μπορεί να παρουσιαστεί είναι το εξής: να πάρουμε όνομα ή επώνυμο που δεν χωράει στον πίνακα που έχουμε προβλέψει. Αλλά, σε τέτοια περίπτωση, δεν υπάρχει λόγος να ρίξουμε εξαίρεση. Απλώς αποθηκεύουμε όσο χωράει.

Η *save()* γράφεται όπως αυτή της *Course*:

```
void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum),
               sizeof(sIdNum) );                          // αριθμός μητρώου
   bout.write( sSurname, sNameSz );
   bout.write( sFirstname, sNameSz );
   bout.write( reinterpret_cast<const char*>(&sNoOfCourses),
               sizeof(sNoOfCourses) );              // αριθ. μαθημ. που δήλωσε
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
                                                         // ώρες ανά εβδομάδα
   if ( bout.fail() )
      throw StudentXptn( "save", StudentXptn::cannotWrite );
} // Student::save
```

Η *display()* μπορεί να είναι χρήσιμη και εδώ:

```
void Student::display( ostream& tout )
{
   tout << sIdNum << '\t' << sSurname << '\t' << sFirstname
        << endl << sNoOfCourses << '\t' << sWH << endl;
} // Student::display
```

Καταλήγουμε λοιπόν στην:

```
class Student
{
public:
   unsigned int getIdNum() const {  return sIdNum;  }
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
   unsigned int getWH() const {  return sWH;  }
   void add1Course( int aIncrWH );
   void readFromText( istream& tin );
   void save( ostream& bout ) const;
   void display( ostream& tout );
private:
   enum { sNameSz = 20 };
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
   unsigned int sWH;           // ώρες ανά εβδομάδα
   unsigned int sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
}; // Student
```

και στην αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentXptn
{
   enum { incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[100];
   int  errorCode;
   int  errValue;
   StudentXptn( char* mn, int ec, int ev = 0 )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errValue = ev;  }
```

```
}; // StudentXptn
```

Και εδώ θα ορίσουμε:

```
typedef Student* PStudent;
```

Prj03.5 Η Κλάση *StudentInCourse*

Για την κλάση αυτήν έχουμε:

- «*Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος) στον πίνακα δηλώσεων.*» Αυτή η απαίτηση μπορεί να αντιμετωπισθεί με έναν δημιουργό, με δύο "*set*" ή με μια "*set*".
- «*Φύλαξε τα στοιχεία του πίνακα δηλώσεων στο* `enrllmnt.dta`.» Από αυτό προκύπτει ανάγκη για μια *save*.

Εδώ, η *save()* είναι εύκολη:

```
void StudentInCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentInCourseXptn( "save", StudentInCourseXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sicSIdNum), sizeof(sicSIdNum) );
                                                          // αριθμός μητρώου
   bout.write( sicCCode, sicCCodeSz );                   // κωδικός μαθήματος
   bout.write( reinterpret_cast<const char*>(&sicMark), sizeof(sicMark) );
                                                          // βαθμός στο μάθημα
   if ( bout.fail() )
      throw StudentInCourseXptn( "save",
                                StudentInCourseXptn::cannotWrite );
} // StudentInCourse::save
```

Ας δούμε τώρα την πρώτη απαίτηση. Για να κάνουμε εισαγωγή ενός ζεύγους στον πίνακα θα πρέπει να έχουμε τη σιγουριά ότι *υπάρχουν και ο φοιτητής και το μάθημα*. Αυτό

- εξασφαλίζεται με αναζήτηση στους αντίστοιχους πίνακες και
- γίνεται με *μια* μέθοδο.

Επιπλέον, αυτή η μέθοδος θα πρέπει να ενημερώνει και τα αντίστοιχα αντικείμενα φοιτητή και μαθήματος.

Άρα οι επιλογές μας περιορίζονται σε δύο: ένας δημιουργός ή μια "*set*". Και οι δύο είναι σωστές, αλλά δεν είναι και τόσο «κομψό» σε δημιουργό μιας κλάσης να περνάς πίνακες με στοιχεία δύο άλλων κλάσεων και να ενημερώνει αντικείμενα άλλων κλάσεων. Για τον λόγο αυτόν επιλέγουμε τη «διπλή» "*set*":

```
void StudentInCourse::setIdNumCCode( unsigned int aIdNum,
                                     string aCCode,
                                     Student stArr[], int nOfStudents,
                                     Course crsArr[], int nOfCourses )
```

Τώρα βέβαια, παρατηρώντας το σχέδιο που κάναμε βλέπουμε ότι όταν έρχεται η ώρα να κάνουμε κάτι τέτοια ξέρουμε ότι ο φοιτητής υπάρχει σίγουρα. Επομένως μπορούμε να περάσουμε στη μέθοδο το αντικείμενο με τα στοιχεία του φοιτητή. Να λοιπόν μια απλοποιημένη μορφή της μεθόδου:

```
void StudentInCourse::setIdNumCCode( Student& aStudent,
                                     string aCCode,
                                     Course crsArr[], int nOfCourses )
{
   crsArr[nOfCourses].setCode( aCCode.c_str() );
   int cPos( 0 );
   while ( strcmp(crsArr[nOfCourses].getCode(),
                  crsArr[cPos].getCode())!=0 )
      ++cPos;
   if ( cPos == nOfCourses )
     throw StudentInCourseXptn( "StudentInCourse",
                                StudentInCourseXptn::unknownCCode,
                                aCCode.c_str() );
```

```
   sicSIdNum = aStudent.getIdNum();
   aStudent.add1Course( crsArr[cPos].getWH() );
   strcpy( sicCCode, aCCode.c_str() );
   crsArr[cPos].add1Student();
} // StudentInCourse::setIdNumCCode
```

Η αναζήτηση μαθήματος με βάση τον κωδικό γίνεται με γραμμική αναζήτηση με φρουρό, αλλά η τοποθέτηση του φρουρού δημιουργεί μια ανάγκη για την κλάση *Course*: χρειαζόμαστε μέθοδο *Course*::*setCode*.

Έτσι, η κλάση γίνεται:

```
class StudentInCourse
{
public:
   void setIdNumCCode( Student& aStudent, string aCCode,
                       Course crsArr[], int nOfCourses );
   void save( ostream& bout ) const;
   void display( ostream& tout ) const;
private:
   unsigned int sicSIdNum;      // αριθμός μητρώου
   char         sicCCode[8];    // κωδικός μαθήματος
}; // StudentInCourse
```

όπου:

```
void StudentInCourse::display( ostream& tout ) const
{
   tout << sicSIdNum << ' ' << sicCCode << endl;
} // StudentInCourse::display
```

Η κλάση εξαιρέσεων είναι:

```
struct StudentInCourseXptn
{
   enum { unknownCCode };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   StudentInCourseXptn( const char* mn, int ec,
                        const char* sv="" )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // StudentInCourseXptn
```

Και εδώ θα ορίσουμε:

```
typedef StudentInCourse* PStudentInCourse;
```

Prj03.6 Το Πρόγραμμα

Ας δούμε τώρα πώς θα υλοποιήσουμε το σχέδιο προγράμματος που κάναμε. Και πρώτα η «*Φόρτωσε τον πίνακα των μαθημάτων.*» Αλλά, πώς θα γίνει η «φόρτωση»; Η φύλαξη έγινε με τη *save* που είδαμε παραπάνω. Αυτό σημαίνει ότι και η φόρτωση θα γίνει με τη *load* (αργότερα θα εξετάσουμε και άλλες δυνατότητες.)

Δηλώνουμε λοιπόν στο πρόγραμμά μας (στη **main**):

```
   Course*      crsArr;
   unsigned int nOfCourses;
```

και στη συνέχεια καλούμε:

```
   loadCourses( "gCourses.dta", crsArr, nOfCourses );
```

Prj03.6.1 Η *loadCourses()*

Τι θα κάνει η *loadCourses()*; Θα πάρει την απαραίτητη μνήμη, θα διαβάσει το αρχείο και θα αποθηκεύσει αυτά που διάβασε στον πίνακα crsArr. Αν κάτι δεν πάει καλά (αν δεν ανοίγει το αρχείο, αν δεν πάρουμε δυναμική μνήμη), θα έχουμε crsArr == 0 και *nOfCourses* == 0 και θα ρίξει εξαίρεση. Δηλαδή:

```
crsArr = 0;   nOfCourses = 0;
Πάρε μνήμη
if ( αποτυχία )  throw ...
Άνοιξε το αρχείο
if ( αποτυχία )
{  επίστρεψε τη μνήμη;  throw ...  }
while ( δεν τελείωσε το αρχείο )
{
   if ( δεν έχεις μνήμη )
      Πάρε και άλλη μνήμη (renew)
   Διάβασε ένα μάθημα
   Μηδένισε το cNoOfStudents του κάθε μαθήματος
   ++nOfCourses;
} // while
```

Στο αρχείο `MyTmpltLib.h` υπάρχει το (γνωστό μας) περίγραμμα

```
// renew - παίρνει μνήμη για πίνακα με nf στοιχεία τύπου T και
//         στα πρώτα ni (<= nf) στοιχεία του αντιγράφουμε αρχικώς
//         αυτά των ni θέσεων του αρχικού p.
template< typename T >
void renew( T*& p, int ni, int nf )
```

και θα το χρησιμοποιήσουμε. Η *renew()* όμως κάνει αντιγραφή του πίνακα κάθε φορά που τον «μεγαλώνει». Για να αποφύγουμε τις πολλές αντιγραφές δεν θα παίρνουμε ένα επιπλέον στοιχείο κάθε φορά αλλά *cIncr* (=10) στοιχεία.[3] Σε μια μεταβλητή, *resCourses*, κρατούμε τον αριθμό στοιχείων που έχουμε δεσμεύσει. Έτσι, η «*Πάρε μνήμη*» γίνεται:

```
   const int cIncr( 10 );
   unsigned int resCourses;    // θέσεις που έχουμε δεσμεύσει
   crsArr = 0;   nOfCourses = 0;
   try {  crsArr = new Course[cIncr];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadCourses", ProgXptn::allocFailed );  }
   resCourses = cIncr;
```

και η «*if (δεν έχεις μνήμη) πάρε και άλλη μνήμη*»:

```
      if ( nOfCourses+1 == resCourses )
      {
         renew( crsArr, nOfCourses, resCourses+cIncr );
         resCourses += cIncr;
      }
```

Εκείνο το "+1", στην *nOfCourses*+1 == *resCourses*, το χρειαζόμαστε για να μπορούμε να κάνουμε γραμμική αναζήτηση με φρουρό.

Να ολόκληρη η *loadCourses()*:

```
void loadCourses( string flNm,
                  Course*& crsArr, unsigned int& nOfCourses )
{
// Πάρε (λίγη) μνήμη
   const int cIncr( 10 );
   unsigned int resCourses;
   crsArr = 0;   nOfCourses = 0;
   try {  crsArr = new Course[cIncr];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadCourses", ProgXptn::allocFailed );  }
   resCourses = cIncr;
// Άνοιξε το αρχείο
```

[3] Αν θέλεις να κάνεις κάτι καλύτερο ξαναδιάβασε αυτά που είπαμε στην §16.13.3.

```
   istream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
   {  delete[] crsArr;   // επίστρεψε τη μνήμη
      throw ProgXptn( "loadCourses", ProgXptn::cannotOpen, flNm.c_str() );
// Τώρα διάβαζε
   loadSylCourse( bin, crsArr[nOfCourses] );
   while ( !bin.eof() )
   {
      ++nOfCourses;
      crsArr[nOfCourses-1].clearStudents();
      if ( nOfCourses+1 == resCourses )
      {
         renew( crsArr, nOfCourses, resCourses+cIncr );
         resCourses += cIncr;
      } // if
      loadSylCourse( bin, crsArr[nOfCourses] );
   } // while
   bin.close();
} // loadCourses
```

Παρατηρήσεις ▶

- Πώς γίνεται το διάβασμα: διαβάζουμε αποθηκεύοντας στο πρώτο στοιχείο μετά το τελευταίο του πίνακα: στο `crsArr[nOfCourses]` (το οποίο υπάρχει πάντοτε). Αν όλα πάνε καλά (αν δεν ρίξει εξαίρεση η *load()*) αυξάνουμε τον *nOfCourses* (`++nOfCourses`) και το νέο στοιχείο περιλαμβάνεται στον πίνακα.
- Αν η *load()* δεν ρίξει εξαίρεση, για κάθε μάθημα που διαβάζουμε, μηδενίζουμε και τον *cNoOfStudents*.
- Η `catch` θα πιάσει οποιαδήποτε *CourseXptn*. Αν ο τύπος του λάθους δεν είναι αυτός που μας ενδιαφέρει (*cannotRead* ή *fileNotOpen*) την ξαναρίχνουμε (`throw`).◀

Prj03.6.2 Η Ανάγνωση του Αρχείου των Δηλώσεων

Η ανάγνωση του αρχείου `enrllmnt.txt` θα «γεμίσει» τον πίνακα φοιτητών και τον πίνακα δηλώσεων φοιτητών σε μαθήματα. Σε μεγάλο ποσοστό θα δουλέψουμε όπως δουλέψαμε παραπάνω. Κατ' αρχήν οι δηλώσεις:

```
   Student*    stArr;   unsigned int nOfStudents( 0 );
   const int sIncr( 10 );
   unsigned int resStudents;

   StudentInCourse* sic; unsigned int nOfStdInCrs( 0 );
   const int sicIncr( 30 );
   unsigned int resStdInCrs;
```

και οι πρώτες «φέτες» δυναμικής μνήμης:

```
      try { stArr = new Student[sIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStudents = sIncr;
      try { sic = new StudentInCourse[scIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStdInCrs = sicIncr;
```

Η «*Άνοιξε το αρχείο enrllmnt.txt*» γίνεται:

```
      ifstream tin( "enrllmnt.txt" );
      if ( tin.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "enrllmnt.txt" );
```

Πριν από αυτό όμως θα πρέπει να ανοίξουμε το αρχείο καταγραφής προβλημάτων (*log*):

```
      log.open( "log.txt" );
      if ( log.fail() )
```

```
        throw ProgXptn( "main",
                        ProgXptn::cannotOpen, "log.txt" );
```

Prj03.6.3 Τα Στοιχεία Ενός Φοιτητή . . .

Η «*Διάβασε τα στοιχεία ενός φοιτητή*» είναι η *readFromText()*; Όχι! Πέρα από τη *readFromText()* θα πρέπει το αντικείμενο που θα διαβαστεί να εισαχθεί στον stArr με ό,τι συνεπάγεται (δυναμική μνήμη.)

```
void readAStudent( istream& tin,
                   PStudent& stArr, unsigned int& nOfStudents,
                   unsigned int& resStudents, int sIncr )
{
   if ( nOfStudents+1 == resStudents )
   {
      renew( stArr, nOfStudents, resStudents+sIncr );
      resStudents += sIncr;
   }
   stArr[nOfStudents].readFromText( tin );
   ++nOfStudents;
   stArr[nOfStudents-1].clearNoOfCourses();
   stArr[nOfStudents-1].clearWH();
} // readAStudent
```

Όπως βλέπεις διαβάζοντας έναν φοιτητή κάνουμε περίπου τα ίδια με αυτά που κάνουμε διαβάζοντας ένα μάθημα. Εδώ όμως υπάρχει μια διαφορά: Υπάρχει πιθανότητα να έχουμε διαβάσει ήδη δήλωση μαθημάτων του φοιτητή. Αυτό

- Γίνεται αντιληπτό από την ύπαρξη αντικειμένου-φοιτητή πιο πριν στον πίνακα μετά από αναζήτηση με τον αριθμό μητρώου.
- Αντιμετωπίζεται με το να αγνοήσουμε την τελευταία δήλωση.

```
        readAStudent( tin,
                      stArr, nOfStudents, resStudents, sIncr );
        if ( !tin.eof() )
        {
           int pos( 0 );
           while ( stArr[pos].getIdNum() !=
                   stArr[nOfStudents-1].getIdNum() ) ++pos;
           if ( pos < nOfStudents-1 )  // υπάρχει ήδη
           {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
```

Τι ακολουθεί για τον φοιτητή και πώς το αγνοούμε;

- Ακολουθεί μια γραμμή με έναν αριθμό, ας τον πούμε *noc*, και
- *noc* γραμμές με έναν κωδικό μαθήματος στην κάθε μια.

Η παρακάτω συνάρτηση τα αγνοεί (τα διαβάζει και τα «ξεχνάει»)):

```
void ignoreStudentData( istream& tin )
{
   string str1;
   getline( tin, str1, '\n' );
   int noc( atoi(str1.c_str()) );
   for ( int k(0); k < noc; ++k )  getline( tin, str1, '\n' );
} // ignoreStudentData
```

Συμπληρώνουμε λοιπόν:

```
           if ( pos < nOfStudents-1 )  // υπάρχει ήδη
           {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
              --nOfStudents;
              ignoreStudentData( tin );
              log << " multiple entry for student with id num "
                  << stArr[pos].getIdNum() << endl;
           }
           else  // δεν υπάρχει
           { . . .
```

Prj03.6.4 . . . Και οι Δηλώσεις Μαθημάτων

Και αν δεν υπάρχει; Ε, τότε ακολουθούμε το σχέδιο:

```
for ( όλα τα μαθήματα που δήλωσε ο φοιτητής )
{
   αναζήτησε τον κωδικό του μαθήματος στον πίνακα μαθημάτων
   if ( δεν υπάρχει )
      Γράψε στο log
   else
   {
      Βάλε το (ΑΜ φοιτητή, κωδικός μαθήματος)
                                          στον πίνακα δηλώσεων
      Ενημέρωσε τα sNoOfCourses και sWH του φοιτητή
      Ενημέρωσε το cNoOfStudents του κάθε μαθήματος
   }
} // for
```

Αυτά τα κάνει η:

```
void readCourseCodes( istream& tin, Student& aStudent,
                      PCourse crsArr, int nOfCourses,
                      PStudentInCourse& sic, unsigned int& nOfStdInCrs,
                      unsigned int& resStdInCrs, int sicIncr,
                      ostream& log )
{
   if ( !tin.eof() )
   {
      string str1;
      getline( tin, str1, '\n' );
      if ( !tin.eof() )
      {
         int noc( atoi(str1.c_str()) );
         for ( int k(0); k < noc; ++k )
         {
            if ( nOfStdInCrs+1 == resStdInCrs )
            {
               renew( sic, nOfStdInCrs, resStdInCrs+sicIncr );
               resStdInCrs += sicIncr;
            }
            getline( tin, str1, '\n' );
            if ( !tin.eof() )
            {
               try
               {
                  sic[nOfStdInCrs].setIdNumCCode( aStudent,
                                                  str1.c_str(),
                                                  crsArr,
                                                  nOfCourses );
                  ++nOfStdInCrs;
               }
               catch( StudentInCourseXptn& x )
               {
                  log << " student with id num "
                      << aStudent.getIdNum()
                      << " asking course " << str1 << endl;
               }
            } // if ( !tin.eof()...
         } // for
      }
   }
} // readCourseCodes
```

Prj03.6.5 Η main

Δες πώς (περίπου) θα είναι η **main**:

```
int main()
```

```
{
   PCourse   crsArr;  unsigned int nOfCourses;

   PStudent  stArr;   unsigned int nOfStudents( 0 );
   const int sIncr( 10 );
   unsigned int resStudents;

   PStudentInCourse sic; unsigned int nOfStdInCrs( 0 );
   const int sicIncr( 30 );
   unsigned int resStdInCrs;

   ofstream log;
   try
   {
      loadCourses( "gCourses.dta", crsArr, nOfCourses );

      try { stArr = new Student[sIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStudents = sIncr;
      try { sic = new StudentInCourse[sicIncr]; }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      resStdInCrs = sicIncr;

      log.open( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt" );
      ifstream tin( "enrllmnt.txt" );
      if ( tin.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "enrllmnt.txt" );
      while ( !tin.eof() )
      {
         readAStudent( tin, stArr, nOfStudents, resStudents, sIncr );
         if ( !tin.eof() )
         {
            int pos( 0 );
            while ( stArr[pos].getIdNum() !=
                    stArr[nOfStudents-1].getIdNum() ) ++pos;
            if ( pos < nOfStudents-1 )  // υπάρχει ήδη
            {// αγνόησε αυτά που ακολουθούν για τον φοιτητή
               --nOfStudents;
               ignoreStudentData( tin );
               log << " multiple entry for student with id num "
                   << stArr[pos].getIdNum() << endl;
            }
            else  // δεν υπάρχει
            {
               readCourseCodes( tin, stArr[nOfStudents-1],
                                crsArr, nOfCourses,
                                sic, nOfStdInCrs, resStdInCrs,
                                sicIncr, log );
            }
         }
         if ( !tin.eof() )
         {
            string str1;
            getline( tin, str1, '\n' ); // blank line
         }
      } // while
      tin.close();
      for ( int k(0); k < nOfStudents; ++k )
      {
         if ( stArr[k].getWH() > 30 )
            log << "student with id num " << stArr[k].getIdNum()
                << ": " << stArr[k].getWH() << " hours/week"
                << endl;
```

```
      } // for
      save1DTable( "Students.dta", stArr, nOfStudents );
      save1DTable( "enrllmnt.dta", sic, nOfStdInCrs );
      save1DTable( "gCourses.dta", crsArr, nOfCourses );
      delete[] sic;
      delete[] stArr;
      delete[] crsArr;
   } // try
   catch( ProgXptn& x )  { . . . } // catch( ProgXptn
   catch( MyLibXptn& x ) { . . . } // catch( MyLibXptn
   catch( CourseXptn& x ) { . . . } // catch( CourseXptn
   catch( StudentXptn& x ) { . . . } // catch( StudentXptn
   catch( StudentInCourseXptn& x )
   { . . . } // catch( StudentInCourseXptn
   catch( ... ) { . . . }
   log.close();
} // main
```

Παρατηρήσεις ▶

- Ο έλεγχος για την υπέρβαση των 30 ωρών ανά εβδομάδα γίνεται (φυσικά) στο τέλος.
- Η *save1DTable()* υπάρχει στο `MyTmpltLib.h`:

```
// save1DTable -- Φυλάγει σε αρχείο binary με όνομα flNm τα n
//                στοιχεία του πίνακα tbl. Πρέπει να έχει ορισθεί
//                η T::save( ostream& )
template < class T >
void save1DTable( string flNm, T tbl[], int n )
{
   if ( tbl == 0 && n > 0 )
      throw MyTmpltLibXptn( "save1DTable",
                            MyTmpltLibXptn::noArray );
   ofstream bout( flNm.c_str(),
                  ios_base::binary|ios_base::trunc );
   if ( bout.fail() )
      throw MyTmpltLibXptn( "save1DTable",
                            MyTmpltLibXptn::cannotCreate,
                            flNm.c_str() );
   for ( int k(0); k < n; ++k )
   {
      tbl[k].save( bout );
      if ( bout.fail() )
         throw MyTmpltLibXptn( "save1DTable",
                               MyTmpltLibXptn::cannotWrite,
                               flNm.c_str() );
   }
   bout.close();
} // save1DTable
```

- Δηλώσαμε το *log* πριν από την `try`, το ανοίξαμε μέσα στη σύνθετη εντολή της `try` και τον κλείνουμε μετά την τελευταία `catch`. Γιατί; Για να έχουμε τη δυνατότητα να γράφουμε στο *log* σε όποιαν `catch` χρειαστεί.
- Πολλές `catch`! Μια για τις εξαιρέσεις του προγράμματος (*ProgXptn*), μια για τις *MyTmpltLibXptn*, μια για τις εξαιρέσεις της κάθε κλάσης και μια για οτιδήποτε άλλο! ◀

Prj03.7 `char*` ή *string*;

Τι γίνεται με τα κείμενα (λεκτικά); Θα τα χειριζόμαστε ως `char*` ή ως *string*; Πριν φθάσουμε στο «δια ταύτα» ξαναδιάβασε την §15.13.1 και ας καταγράψουμε μερικά βασικά σημεία.

Γενικώς, *ο τύπος string είναι βολικότερος και τα προγράμματά μας γράφονται πιο εύκολα. Αλλά, το κείμενο αποθηκεύεται σε δυναμική μνήμη.* Συνέπειες σε αυτά που είδαμε μέχρι τώρα:

- Αν στην κλάση εξαιρέσεων δηλώσουμε `string funcName` πρόσεξε τι μπορεί να συμβεί: Προσπαθείς ανεπιτυχώς να πάρεις δυναμική μνήμη και θέλεις να ρίξεις σχετική εξαίρεση. Αφού στην εξαίρεση θα χρειαστεί δυναμική μνήμη το πιο πιθανό είναι να ριχτεί (καινούρια) `std::bad_alloc`. Φυσικά, τέτοια εξαίρεση θα μπορεί να ριχτεί και κατά την (όποια) αντιγραφή της εξαίρεσης και –όπως θα μάθουμε αργότερα– να διακοπεί η εκτέλεση του προγράμματός σου.
- Αν στη *Student* είχαμε δηλώσει

```
string  sSurname;
string  sFirstname;
```

πώς θα κάναμε τη φύλαξη αυτού του μέλους στη *save()*; Θα κάναμε κάτι σαν:

```
char    local[sNameSz];
strncpy( local, sSurname.c_str(), sNameSz-1 );  local[sNameSz-1] = '\0';
bout.write( local, sNameSz );
strncpy( local, sFirstname.c_str(), sNameSz-1 );  local[sNameSz-1] = '\0';
bout.write( local, sNameSz-1 );
```

Δεν σου αρέσει; Στην §15.13.1 λέγαμε: «*Δεν θα μπορούσαμε να φυλάγουμε τις τιμές των ... μελών τύπου string με το ακριβές περιεχόμενο που έχουν κάθε φορά οποιοδήποτε μήκος και αν έχει; Ναι, αρκεί να φυλάγουμε και την τιμή του (κάθε) μήκους.*» Δες λοιπόν και αυτό:

```
unsigned int len( sSurname.length() );
char*        local( new char[len+1] );
strcpy( local, sSurname,c_str() );
bout.write( reinterpret_cast<const char*>(&len), sizeof(len) );
bout.write( local, len+1 );
delete[] local;
len = sFirstname,length();
local = new char[len+1];
strcpy( local, sFirstname );
bout.write( reinterpret_cast<const char*>(&len), sizeof(len) );
bout.write( local, len+1 );
```

Εδώ έχεις αυτό που λέμε «εγγραφές μεταβλητού μήκους».

Είναι φανερό ότι η λύση που έχουμε επιλέξει είναι απλούστερη και από τους δύο τρόπους που είδαμε εδώ.

Φυσικά τίποτε δεν μας εμποδίζει

- να έχουμε τα μέλη των κλάσεων δηλωμένα ως πίνακες `char` αλλά
- να κάνουμε όλους τους άλλους χειρισμούς στον τύπο *string*.

Αυτό κάναμε παραπάνω και έτσι θα συνεχίσουμε. Αλλά, όπως θα δούμε στο επόμενο κεφάλαιο, υπάρχει πιθανότητα –κατ' αρχήν τουλάχιστον– στις αντιγραφές τιμών τύπου *string* να εγερθεί εξαίρεση *bad_alloc*![4]

Prj03.8 Ένα Άλλο Σχέδιο για τις Κλάσεις

Η ιδέα να έχουμε στη *StudentInCourse* μια μέθοδο σαν την

```
void setIdNumCCode( Student& aStudent, string aCCode,
                    Course crsArr[], int nOfCourses );
```

ή, εναλλακτικώς, έναν δημιουργό σαν τον:

```
StudentInCourse::StudentInCourse( Student& aStudent, string aCCode,
                                  Course crsArr[], int nOfCourses )
// . . .
```

δεν είναι και τόσο «ορθόδοξη».

[4] Είπαμε: «*κατ' αρχήν*». Αν είναι να ψάχνουμε για εξαιρέσεις κάθε φορά που κάνουμε κάτι με τιμές τύπου *string*…

Θα έλεγε κανείς ότι καλύτερο θα ήταν να δηλώσουμε τον πίνακα crsArr *μέσα* στη *StudentInCourse*. Δηλαδή θα έχουμε αυτόν τον πίνακακ σε κάθε αντικείμενο *StudentInCourse*; Θα δούμε στα επόμενα μαθήματα ότι υπάρχει τρόπος να το αποφύγουμε.

Πάντως, ούτε αυτό είναι καλό αφού ένας τέτοιος πίνακας πιθανόν να μας χρειάζεται και για άλλες χρήσεις.

Ο συνήθης τρόπος χειρισμού αυτής της κατάστασης είναι να *έχουμε τον πίνακα* crsArr *έξω από την κλάση και μέσα στην κλάση StudentInCourse να έχουμε ένα βέλος προς αυτόν*:

```
class StudentInCourse
{
public:
// . . .
private:
   Course*       crsArr;
   unsigned int* pNOfCourses;
   unsigned int  sicSIdNum;      // αριθμός μητρώου
   char          sicCCode[8];    // κωδικός μαθήματος
}; // StudentInCourse
```

Αλλα το βέλος προς τον πίνακα δεν είναι σταθερό αφού αλλάζει κάθε τόσο με τη *renew()*. Θα το ξαναδούμε αργότερα...

Ερωτήσεις – Ασκήσεις

Α Ομάδα

Prj03-1 Ας πούμε ότι ο κωδικός μαθήματος έχει την εξής δομή:

- Δύο κεφαλαία γράμματα του ελληνικού αλφαβήτου που μπορεί να είναι "`ΕΥ`" ή "`ΤΠ`".
- Τέσσερα ψηφία του δεκαδικού συστήματος.
- Ένα κεφαλαίο γράμμα του ελληνικού αλφαβήτου που μπορεί να είναι "`Θ`" ή "`Ε`".

Κάνε τις απαραίτητες συμπληρώσεις/διορθώσεις σε αυτά που γράψαμε.

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:

20.1 Άλλο Ένα Παράδειγμα: *BString* 658
- 20.1.1 Οι Απλές Μέθοδοι 659
- 20.1.2 … Και η Μέθοδος *at()* 661
- 20.1.3 Ο Καταστροφέας 662

20.2 Ο Δημιουργός Αντιγραφής 662

20.3 Η Πρόσβαση στα Μέλη "`private`" 664

20.4 Ο Τελεστής Εκχώρησης "=" 664
- 20.4.1 Η Μέθοδος *assign()* 666
- 20.4.2 Και Μια Εκχώρηση που *δεν* Ορίσαμε 667

20.5 Το Βέλος "`this`" 668

20.6 Επιστρέφουμε Τύπο Αναφοράς; 669

20.7 Μια Κλάση για Διαδρομές Λεωφορείων 669
- 20.7.1 Η Κλάση για τις Στάσεις 670
- 20.7.2 Η Κλάση για τις Διαδρομές 671
- 20.7.3 Οι Κλάσεις Τελικώς 679
- 20.7.4 Και το Πρόγραμμα 681
- 20.7.5 Σχόλια, Παρατηρήσεις κλπ 683

20.8 Συσχετίσεις Κλάσεων 687
- 20.8.1 Διαγραμματικές Παραστάσεις 691

20.9 Για να Γράψουμε μια Κλάση... 694

Ερωτήσεις - Ασκήσεις 695
- Α Ομάδα 695
- Β Ομάδα 695

Γ Ομάδα 695

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.

---

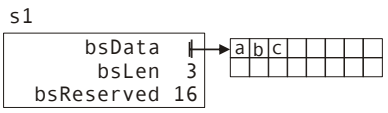
[1] Ξαναδές και αυτά που είπαμε στο Παράδ. 2 της §16.13.

[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:

$$(0 \le bsLen < bsReserved) \;\&\&\; (bsReserved \;\%\; bsIncr == 0)$$

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: ▶

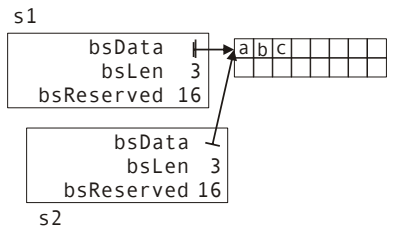
Αν προτιμάς να γράφεις

```
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;
```



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

Αποτέλεσμα:

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

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;
```

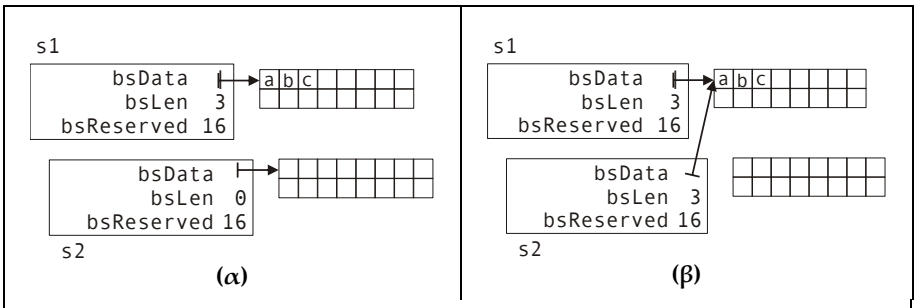
Αποτέλεσμα:

```
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"

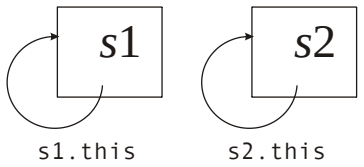
Στην προηγούμενη παράγραφο χρειαστήκαμε, μέσα σε μια μέθοδο, βέλος προς το αντικείμενο-ιδιοκτήτη της μεθόδου. Πώς το βρίσκουμε αυτό; Ένας τρόπος θα ήταν να πάρουμε ένα βέλος προς το πρώτο μέλος του αντικειμένου, αφού, όπως ξέρουμε από τις δομές, τα δύο βέλη είναι ίσα. Φυσικά, για να μπορέσουμε να το 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.***



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.

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;
```

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* ≥ 0 και τη συνθήκη διαχείρισης του δυναμικού πίνακα:

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

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:

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

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:

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

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*:

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

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

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

όπου (προσοχή στις αλλαγές των δεικτών!)

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

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

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*).

$$
\begin{aligned}
I_I \equiv\ & (0 \leq rNoOfStops \leq rReserved\text{-}2)\ \&\&\ (rReserved\ \%\ rIncr == 0)\ \&\& \\
& (\forall\ k\colon 1..rNoOfStops\text{-}1 \bullet (\text{rAllStops}[k+1].sDist > \text{rAllStops}[k].sDist)\ \&\& \\
& (\text{rAllStops}[0].sDist == 0\ \&\&\ \text{rAllStops}[0].sFare == 0)\ \&\& \\
& (\text{rAllStops}[rNoOfStops+1].sDist == FLT_MAX\ \&\& \\
& \text{rAllStops}[rNoOfStops+1].sFare == FLT_MAX)
\end{aligned}
$$

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

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

προφανώς ισχύει και το τμήμα της I_D

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

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 *"ΜΑΡΜΑΡΑ"* θα γίνει ως εξής:

```
      if ( oneRoute.find1RouteStop("ΜΑΡΜΑΡΑ") )
         cout << "Στάση ΜΑΡΜΑΡΑ υπάρχει" << endl;
      else
         oneRoute.addRouteStop( RouteStop("ΜΑΡΜΑΡΑ", 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: $(a == b) \equiv (!(a < b) \;\&\&\; !(b < a))$.

```
} // 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 περιεχομενων 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; // εισπρακτορας
}; // 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.

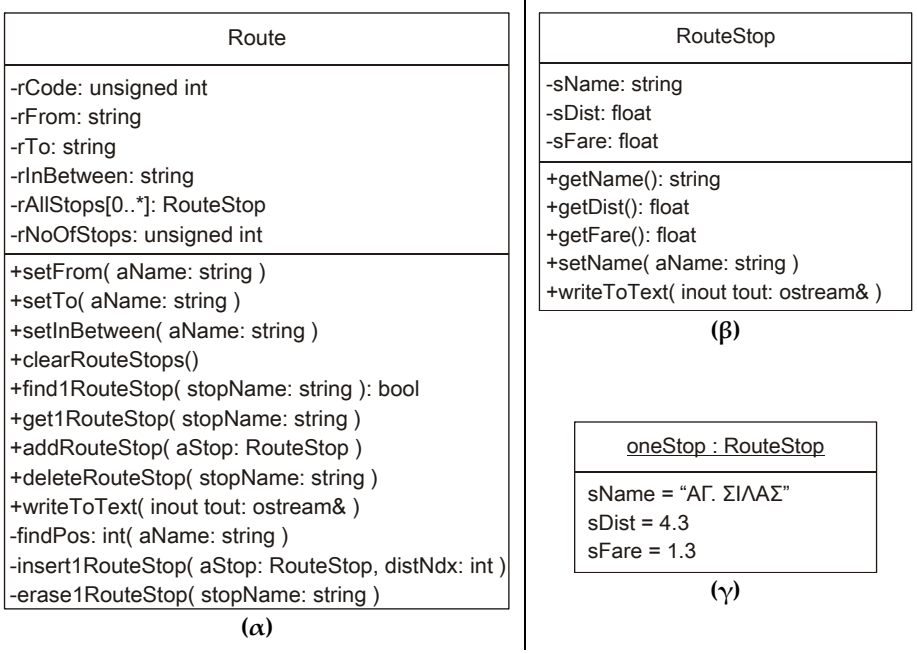
Οι συσχετίσεις που είδαμε παραπάνω είναι διμελείς και απλές. Στη φάση της σχεδίασης οι συσχετίσεις μπορεί να είναι πολυμελείς και πιο πολύπλοκες. Μια τέτοια συσχέτιση μπορεί να συνοδεύεται και από αντίστοιχη **κλάση συσχέτισης** (association class).

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

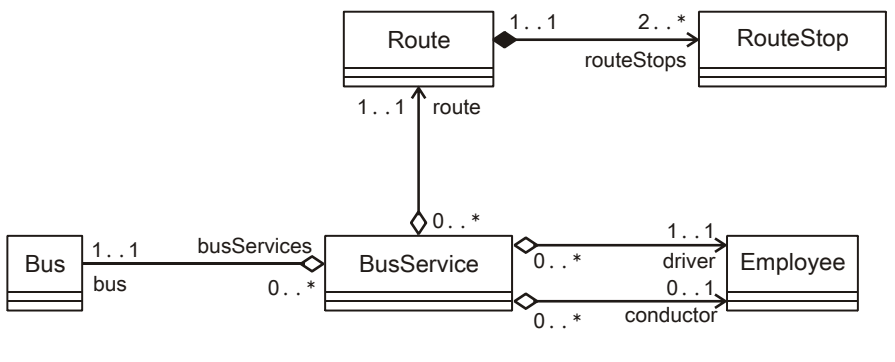
```
class BusStop
{
public:
// . . .
private:
   string        bsName;      // όνομα στάσης
   string        bsLocation;  // θέση
   unsigned char bsFaciLevel; // επίπεδο εξυπηρέτησης
}; // BusStop
```

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`";

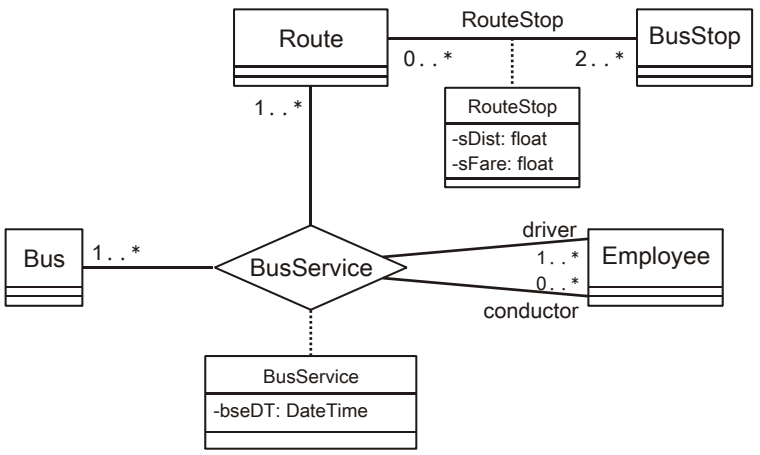
Αν δεν υπάρχει το *είδος* εννοείται "`in`".

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.

κεφάλαιο

21

Ειδικές Συναρτήσεις και Άλλα

Ο στόχος μας σε αυτό το κεφάλαιο:
Να γνωρίσουμε καλύτερα μερικά βασικά εργαλεία που υπάρχουν σε κάθε κλάση (είτε το θέλουμε είτε όχι). Τελικώς θα καταλήξουμε σε μια «συνταγή» που θα πρέπει να ακολουθούμε όταν σχεδιάζουμε οποιαδήποτε κλάση: Τι πρέπει να σκεφθούμε και να αποφασίσουμε ασχέτως από τις εφαρμογές όπου θα χρησιμοποιηθεί η κλάση μας.

Προσδοκώμενα αποτελέσματα:
Όταν θα έχεις μελετήσει αυτό το τεύχος θα μπορείς να σχεδιάσεις και να υλοποιήσεις, κατ' αρχήν, οποιαδήποτε κλάση.

Έννοιες κλειδιά:

- *δημιουργός*
- *ερήμην δημιουργός*
- *δημιουργός αντιγραφής*
- *λίστα εκκίνησης*
- *καταστροφέας*
- *τελεστής εκχώρησης*
- *RAII*
- *στατικά μέλη*
- *σταθερά μέλη*

Περιεχόμενα:

21.1 Ερήμην Δημιουργός 698
21.1.1 Δημιουργός με Αρχική Τιμή 700
21.2 Δημιουργός Αντιγραφής 701
21.3 Σειρά Δημιουργίας – Λίστα Εκκίνησης 704
21.4 Εξαιρέσεις από τον Δημιουργό 706
21.5 Ο Καταστροφέας 708
21.5.1 Ο Καταστροφέας *δεν* Ρίχνει Εξαιρέσεις 711
21.5.2 Καλούμε τον Καταστροφέα; 711
21.6 Ο Τελεστής Εκχώρησης 712
21.6.1 Η Ασφαλής *swap()* 712
21.7 * Προσωρινά Αντικείμενα 716
21.8 Ο «Κανόνας των Τριών» 717
21.9 Μια Παρένθεση για τη *renew()* 717
21.10 Αυτά που Μάθαμε στην Πράξη: ΑΝ ΔΕ {ΔΑ ΤΕ ΚΑ} GE SE 719
21.10.1 * Επιστροφή στις *string* και *BString*: Μέθοδος *reserve()* 721
21.11 Λίστα με Απλή Σύνδεση 723
21.11.1 Άλλες Μέθοδοι; 728
21.11.2 Το Πρόγραμμα 729
21.12 * Βέλος προς Μέθοδο 730

21.13 Μετατροπές Τύπου ... 731
21.13.1 Μετατροπή με Δημιουργό ... 731
21.13.2 Συναρτήσεις Μετατροπής ... 733
21.14 Στατικά Μέλη Κλάσης ... 734
21.15 «Σταθερά» Μέλη Κλάσης ... 736
21.16 «Σταθερά» Μέλη Αντικειμένου ... 737
Ερωτήσεις - Ασκήσεις ... 738
Α Ομάδα ... 738
Β Ομάδα ... 738

Εισαγωγικές Παρατηρήσεις:

Όπως λέγαμε στο εισαγωγικό σημείωμα του προηγούμενου κεφαλαίου, στον ορισμό μιας κλάσης «*Περιγράφεται επίσης και η λειτουργία* ***δημιουργίας στιγμιοτύπου*** *(instance creation).*» Στην πραγματικότητα η λειτουργία δημιουργίας στιγμιοτύπου είναι το εργαλείο που μας χρειάζεται για να διασφαλίσουμε ότι η αρχική κατάσταση του αντικειμένου συμμορφώνεται με την αναλλοίωτη. Στη C++ μας δίνεται η δυνατότητα να καθορίσουμε εμείς το πώς μπορεί να γίνονται οι δηλώσεις των αντικειμένων μιας κλάσης, π.χ. μεταβλητών τύπου *Date*. Αυτό γίνεται με τον ορισμό ειδικών μεθόδων που λέγονται **δημιουργοί** (creators) ή **κατασκευαστές** (constructors).

Μέχρι τώρα είδαμε δημιουργούς για τις κλάσεις *Date*, *BString*, *Battery*, *Route* και *RouteStop*. Στη *BString* και στη *Route* παρουσιάστηκε και άλλη ανάγκη: κάθε αντικείμενό τους δεσμεύει πόρους του συστήματος (μνήμη) που πρέπει να αποδεσμευθεί όταν το αντικείμενο καταστρέφεται. Έτσι, υποχρεωθήκαμε να ορίσουμε και **καταστροφέα** (destructor) όπου περιγράφουμε πώς αποδεσμεύονται οι πόροι (μνήμη) που δεσμεύθηκαν.

Ενώ όμως για κάθε κλάση έχουμε έναν καταστροφέα, συχνά χρειαζόμαστε περισσότερους από έναν δημιουργούς για διαφορετικές χρήσεις. Στη συνέχεια θα δούμε τις διάφορες περιπτώσεις όπου καλούνται οι δημιουργοί.

Τέσσερα από αυτά τα εργαλεία ονομάζονται **ειδικές συναρτήσεις** (special functions): ο *ερήμην δημιουργός*, ο *δημιουργός αντιγραφής*, ο *καταστροφέας* και ο *τελεστής εκχώρησης* και μοιράζονται μια άλλη πολύ ενδιαφέρουσα ιδιότητα: αν δεν τα ορίσουμε εμείς ορίζει αυτόματως (και ερήμην μας) ο μεταγλωττιστής **συναγόμενες** (implicit) αντίστοιχες συναρτήσεις. Είναι σωστές; Πολύ συχνά όχι. Είδαμε ήδη παραδείγματα για την κλάση *BString*.

Πολλοί περιλαμβάνουν στις ειδικές συναρτήσεις όλους τους δημιουργούς καθώς και τις **συναρτήσεις μετατροπής τύπου** (conversion functions) που θα δούμε επίσης.

Δύο προβλήματα που αντιμετωπίζουμε συχνά γράφοντας δημιουργούς –και άλλες μεθόδους– είναι:

- Η ανάγκη για σταθερές με όνομα, για να αποφύγουμε τις λεγόμενες «μαγικές σταθερές».
- Η ανάγκη για βοηθητικές συναρτήσεις.

Εδώ θα τα δούμε πληρέστερα.

Τέλος, θα δούμε τις περιπλοκές από τη χρήση εξαιρέσεων στους δημιουργούς και τους καταστροφείς και που χρειάζονται ιδιαίτερη προσοχή.

Στη τέλος του κεφαλαίου θα βρεις τη «συνταγή»-ανακεφαλαίωση.

21.1 Ερήμην Δημιουργός

Ο **ερήμην δημιουργός** (default constructor) είναι αυτός που δημιουργεί ένα αντικείμενο και καθορίζει την κατάστασή του (τιμές των μελών) αυτομάτως, ερήμην του προγραμματιστή, δηλαδή *ο δημιουργός που μπορεί να κληθεί χωρίς παραμέτρους*. Η πιο απλή περίπτωση τέτοιου δημιουργού είναι αυτός που δεν έχει παραμέτρους.

Για να τον δούμε καλύτερα, αλλάζουμε λιγάκι τη *Date*:

```
class Date
{ // I: (dYear > 0) && (0 < dMonth <= 12) &&
  //    (0 < dDay <= lastDay(dYear, dMonth))
public:
//   Date( int yp = 1, int mp = 1, int dp = 1 );
   Date();
   Date( int yp, int mp = 1, int dp = 1 );
// . . .
```

δηλαδή, απομονώνουμε την περίπτωση:

```
Date::Date()
{
   cout << "in Date default constructor" << endl;
   dYear = 1;  dMonth = 1;  dDay = 1;
                                  // 1η Ιανουαρίου του έτους 1 μ.Χ.
} // Date::Date
```

Οι εντολές:

```
   cout << "απλή δήλωση" << endl;
   Date d;
   cout << "δήλωση πίνακα" << endl;
   Date da[3];
   cout << "δυναμική μεταβλητή" << endl;
   Date* pd( new Date );
   cout << "δυναμικός πίνακας" << endl;
   Date* pda( new Date[2] );
```

θα μα ς δώσουν:

```
απλή δήλωση
in Date default constructor
δήλωση πίνακα
in Date default constructor
in Date default constructor
in Date default constructor
δυναμική μεταβλητή
in Date default constructor
δυναμικός πίνακας
in Date default constructor
in Date default constructor
```

Έτσι, εμπειρικώς, βλέπουμε ότι ο ερήμην δημιουργός καλείται όταν:

- Δηλώνουμε μια μεταβλητή χωρίς αρχική τιμή.
- Δηλώνουμε σταθερό πίνακα (μια φορά για κάθε στοιχείο).
- Παίρνουμε δυναμική μνήμη για μια μεταβλητή χωρίς αρχική τιμή.
- Παίρνουμε δυναμική μνήμη για πίνακα (μια φορά για κάθε στοιχείο).

Καταλαβαίνεις ότι σπανίως θα γράψεις κλάση χωρίς ερήμην δημιουργό.

Παρατηρήσεις: ▶

1. Γιατί στον δημιουργό με αρχική τιμή γράψαμε:

```
   Date( int yp, int mp = 1, int dp = 1 );
```

και όχι

```
   Date( int yp = 1, int mp = 1, int dp = 1 );
```

Διότι ο μεταγλωττιστής δεν θα ήξερε ποιον από τους δύο δημιουργούς να διαλέξει σε όλες τις δηλώσεις του παραδείγματος, αφού και οι δύο θα ήταν κατάλληλοι.

2. Δες έναν τρόπο αντιμετώπισης των «μαγικών σταθερών»:

```
   dYear = 1;  dMonth = 1;  dDay = 1;        // 1η Ιανουαρίου του έτους 1 μ.Χ.
```

Καλά, με σχόλιο; Ε, με σχόλιο, τι να κάνουμε; . . .◀

21.1.1 Δημιουργός με Αρχική Τιμή

Όπως είπαμε παραπάνω «*Ο ερήμην δημιουργός (default constructor) είναι ... ο δημιουργός που μπορεί να κληθεί χωρίς παραμέτρους.*» Τέτοιος δημιουργός είναι και αυτός που ήδη ονομάσαμε «2 σε 1» (ή «πολλοί σε 1»): *αυτός που έχει παραμέτρους αλλά όλες έχουν ερήμην καθοριζόμενες τιμές*. Αυτός είναι συνήθως πιο πολύπλοκος από τον ερήμην δημιουργό χωρίς παραμέτρους αφού θα πρέπει να κάνουμε ελέγχους συμμόρφωσης με την αναλλοίωτη. Και τι γίνεται αν βρούμε παρανομίες; Ρίχνουμε εξαιρέσεις!

Για παράδειγμα στη *Date*, δηλώνουμε (§19.1):

```
   Date( int yp = 1, int mp = 1, int dp = 1 );
```

ορίζουμε:

```
Date::Date( int yp, int mp, int dp )
{
cout << "in Date default constructor: ";
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   dYear = yp;
   if ( mp <= 0 || 12 < mp )
      throw DateXptn( "Date", DateXptn::monthErr, mp );
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
cout << dDay << '.' << dMonth << '.' << dYear << endl;
} // Date
```

και οι εντολές:

```
   cout << "απλή δήλωση" << endl;
   Date d;
   cout << "δήλωση πίνακα" << endl;
   Date da[3];
   cout << "δυναμική μεταβλητή" << endl;
   Date* pd( new Date );
   cout << "δυναμικός πίνακας" << endl;
   Date* pda( new Date[2] );
```

θα μα ς δώσουν:

```
απλή δήλωση
in Date default constructor: 1.1.1
δήλωση πίνακα
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
δυναμική μεταβλητή
in Date default constructor: 1.1.1
δυναμικός πίνακας
in Date default constructor: 1.1.1
in Date default constructor: 1.1.1
```

Αυτός ο δημιουργός μπορεί να λειτουργεί ως ερήμην δημιουργός αλλά και ως **δημιουργός με αρχική τιμή** (initializing constructor). Δοκιμάζουμε τις εντολές:

```
   cout << "δήλωση με αρχική τιμή" << endl;
   Date d( 2006, 8, 26 );
   cout << "δήμιουργία σταθεράς" << endl;
   Date( 2007, 10, 5 );
   cout << "δυναμική μεταβλητή με αρχική τιμή" << endl;
   Date* pd( new Date(2001, 1, 11) );
   cout << "μετατροπή τύπου ορίσματος" << endl;
   bool before( d < 2008 );
   cout << before << endl;
```

Αποτέλεσμα:

```
δήλωση με αρχική τιμή
in Date default constructor: 26.8.2006
```

```
δήμιουργία σταθεράς
in Date default constructor: 5.10.2007
δυναμική μεταβλητή με αρχική τιμή
in Date default constructor: 11.1.2001
μετατροπή τύπου ορίσματος
in Date default constructor: 1.1.2008
1
```

Βλέπουμε λοιπόν ότι ο δημιουργός με αρχική τιμή καλείται όταν:

- Δηλώνουμε μια μεταβλητή *με αρχική τιμή*.
- Δημιουργούμε μια «σταθερά της κλάσης».
- Παίρνουμε δυναμική μνήμη για μια μεταβλητή *με αρχική τιμή*.
- Περνούμε σε παράμετρο μια τιμή (2008) που πρέπει να *μετατραπεί* σε τιμή της κλάσης μας (*Date*).

Ερήμην δημιουργό με αρχική τιμή έχουμε και στην κλάση *BString*, όπου δηλώνουμε:

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

και ορίζουμε:

```
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
```

Υπάρχει περίπτωση να έχουμε δημιουργό που χρειάζεται οπωσδήποτε κάποια αρχική τιμή στη δήλωση; Δεν είναι τόσο συχνή περίπτωση αλλά υπάρχει πιθανότητα να χρειαστείς κάτι τέτοιο. Ο δημιουργός της *Date* μπορεί να γίνει έτσι αν τον δηλώσεις:

```
   Date( int yp, int mp = 1, int dp = 1 );
```

και ο δημιουργός της *BString* αν τον δηλώσεις:

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

21.2 Δημιουργός Αντιγραφής

Μια ειδική κατηγορία δημιουργών είναι οι **δημιουργοί αντιγραφής** (copy constructors). Αυτοί καλούνται όταν κατά τη δήλωση ενός αντικειμένου του δίνουμε ως αρχική τιμή την τιμή ενός άλλου της ίδιας κλάσης, π.χ.:

```
   Date d1( 2002, 7, 2 ), d2( d1 );  // d1 == d2 == 02.07.2002
```

ή

```
   Date d1( 2002, 7, 2 ), d2 = d1;   // d1 == d2 == 02.07.2002
```

Στην περίπτωση αυτή καλείται ο ερήμην καθορισμένος δημιουργός αντιγραφής της κλάσης. Αυτός δουλεύει ως εξής: αντιγράφει, μια προς μια, τις τιμές όλων των μελών του *d1* στα μέλη του *d2*.

Αυτό δεν είναι πάντοτε ικανοποιητικό και συχνά πρέπει να γράψουμε δικό μας δημιουργό αντιγραφής. Στην §20.2 είδαμε ένα τέτοιο παράδειγμα στην κλάση *BString*, για την οποίαν γράψαμε τον δημιουργό αντιγραφής:

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

```
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BString
```

Πότε γράφουμε δημιουργό αντιγραφής;

- ***Γράφουμε δημιουργό αντιγραφής όταν η αντιγραφή μέλος προς μέλος που θα κάνει αυτομάτως η C++ δεν κάνει αυτό που θέλουμε.***

Αυτό είναι πολύ γενικό· γιατί να μην πούμε «όταν τα αντικείμενά της κλάσης παίρνουν δυναμική μνήμη»; Αυτή είναι μια ειδική περίπτωση *δέσμευσης πόρων του συστήματος από αντικείμενα της κλάσης*. Και αυτή η διατύπωση όμως δεν καλύπτει τα πάντα.

- Αργότερα θα δούμε παράδειγμα όπου ο δημιουργός αντιγραφής δεν θα κάνει αντιγραφή αλλά *μεταβίβαση*.
- Υπάρχουν περιπτώσεις που θέλουμε απλώς να «αχρηστεύσουμε» τον δημιουργό αντιγραφής που γράφεται αυτομάτως.

Από τις κλάσεις που είδαμε μέχρι τώρα:

- Για τις *Date*, *GrElmn* και *RouteStop* δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής.
- Χρειάζεται όμως για τη *BString* και τη *Route* διότι οι τιμές των αντικειμένων αυτών τών κλάσεων δεν υπάρχουν εξ ολοκλήρου στις τιμές των μελών αλλά σε δυναμική μνήμη που δεν αντιγράφεται στην αντιγραφή μέλος προς μέλος.

Τώρα πρόσεξε το εξής:

- ***Η παράμετρος του δημιουργού αντιγραφής πρέπει να είναι παράμετρος αναφοράς. Κάθε δημιουργός αντιγραφής μιας κλάσης K θα δηλώνεται με επικεφαλίδα: "`K( const K& rhs )`".***

Στη συνέχεια θα καταλάβεις το γιατί.

Μπορεί ο δημιουργός αντιγραφής να να έχει και άλλες παραμέτρους; Ναι, αλλά θα πρέπει να εμφανίζονται μετά από την "`const K& rhs`" και *να έχουν όλες ερήμην καθορισμένες τιμές*.

Με τον δημιουργό αντιγραφής η C++ κάνει δύο ακόμη πολύ σημαντικές δουλειές:

- Πέρασμα παραμέτρων τιμής σε συνάρτηση.
- Επιστροφή τιμής (εκτέλεση της `return`) από συνάρτηση με τύπο.

Ας πούμε ότι έχουμε μια συνάρτηση:[1]

```
BString appendW( BString s )
{
   BString fv;
   char* sv( new char[s.length()+2] );
   strcpy( sv, s.c_str() );
   sv[s.length()] = 'W';  sv[s.length()+1] = '\0';
   fv = BString( sv );
cout << "returning from appendW" << endl;
   return fv;
} // appendW
```

Αλλάζουμε λίγο τους δημιουργούς της *BString*. Στον ερήμην δημιουργό βάζουμε:

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

και στον δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
cout << "In copy constructor; rhs: " << rhs.c_str() << endl;
   try {  bsData = new char [bsReserved];  }
```

[1] Τι κάνει αυτή η (άχρηστη!) συνάρτηση;

```
// . . .
```

Σε ένα πρόγραμμα καλούμε την *appendW()* ως εξής:

```
      BString q( "abc" ), t;
cout << "going to call appendW" << endl;
      t = appendW( q );
      cout << "q: " << q.c_str() << "   t: " << t.c_str() << endl;
```

και παίρνουμε:[2]

```
In default constructor; rhs: abc
In default constructor; rhs:
going to call appendW
In copy constructor; rhs: abc
In default constructor; rhs:
In default constructor; rhs: abcW
returning from appendW
In copy constructor; rhs: abcW
q: abc   t: abcW
```

Αμέσως μετά το μήνυμα «`going to call appendW`» βλέπεις το «`In copy constructor; rhs: abc`». Παρομοίως, αμέσως μετά το «`returning from appendW`» βλέπεις το «`In copy constructor; rhs: abcW`».

Στην §7.3 μάθαμε για τη κλήση συνάρτησης με παραμέτρους τιμής: «*Κατ' αρχήν παραχωρείται στη συνάρτηση ... μνήμη για όλα τα τοπικά αντικείμενα: [τυπικές παραμέτρους τιμής και τοπικές μεταβλητές]. Στη συνέχεια αντιγράφονται οι τιμές των πραγματικών παραμετρων στις αντίστοιχες τυπικές.*» Αυτό το «*στη συνέχεια αντιγράφονται οι τιμές*» μάλλον σε κάνει να σκεφτείς τον τελεστή εκχώρησης αλλά δεν είναι έτσι. Το μήνυμα «`In copy constructor; rhs: abc`» αμέσως μετά την κλήση της συνάρτησης μας δείχνει ότι *δημιουργείται ένα τοπικό αντικείμενο s με αρχική τιμή την τιμή της q*. Δηλαδή, όταν καλείται η συνάρτηση είναι σαν να εκτελείται η δήλωση:

```
BString s( q );
```

Στη συνέχεια, λέγαμε στην §7.3 «*Η τιμή της παράστασης που υπάρχει στη* `return` *... αντικαθιστά την κλήση της συνάρτησης.*» Στην περίπτωσή μας η *fv* που υπάρχει στη `return` είναι μια μεταβλητή τοπική της *appendW()*, που θα χαθεί όταν τελειώσει η εκτέλεση της συνάρτησης. Έτσι, δημιουργείται μια προσωρινή θέση στη μνήμη, ας την πούμε *tmp*, όπου αντιγράφεται η τιμή της *fv*. Και αυτό γίνεται με τον δημιουργό αντιγραφής· είναι σαν να εκτελείται η δήλωση:

```
BString tmp( fv );
```

και από αυτήν παίρνουμε το μήνυμα «`In copy constructor; rhs: abcW`».

Σημείωση: ▶
Αν αντί για "`return fv`" γράψεις "`return BString(sv)`" ή "`return sv`"το πιο πιθανό είναι να κληθεί μόνον ο ερήμην δημιουργός (με αρχική τιμή). Παρομοίως, θα κληθεί ο δημιουργός με αρχική τιμή αν καλέσεις την *appendW* με όρισμα `"abc"`. ◀

Θα πρέπει λοιπόν να θυμάσαι ότι:

- ***Το πέρασμα της τιμής της πραγματικής παραμέτρου στην τυπική παράμετρο τιμής, στη C++, γίνεται με τον τρόπο που δίνουμε αρχική τιμή σε μια μεταβλητή –δηλαδή με τον δημιουργό αντιγραφής– και όχι με εκχώρηση.***
- ***Η επιστροφή τιμής μιας συνάρτησης –με την εντολή*** `return`***– στη C++, γίνεται με τον τρόπο που δίνουμε αρχική τιμή σε μια μεταβλητή: με τον δημιουργό αντιγραφής.***

Και τώρα ας ξανασκεφτούμε γιατί στον δημιουργό αντιγραφής περνούμε το προς αντιγραφή αντικείμενο *υποχρεωτικώς* με παράμετρο αναφοράς και όχι τιμής. Αν βάζαμε παράμετρο τιμής *θα χρησιμοποιούσαμε στον ορισμό του δημιουργού αντιγραφής τον δημιουργό*

[2] Borland C++, v.5.5

αντιγραφής (που θα ενεργοποιηθεί για το πέρασμα παραμέτρου τιμής), θα είχαμε δηλαδή μια αέναη αναδρομή!

21.3 Σειρά Δημιουργίας – Λίστα Εκκίνησης

Ας δούμε τώρα με ποια σειρά γίνεται η δημιουργία ενός αντικειμένου[3].

- Δημιουργούνται τα μέλη με τη σειρά που δηλώνονται.
- Εκτελείται το σώμα του δημιουργού.

  Αυτό φαίνεται και στο παρακάτω

Παράδειγμα 1 ✎

Το πρόγραμμα:

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class A
 4: {
 5:    int aM;
 6: public:
 7:    A( int ap = 0 )    // ερήμην δημιουργός
 8:    {  aM = ap;
 9:       cout << "A object created, aM = " << aM << endl;  }
10:    A( const A& rhs )  // δημιουργός αντιγραφής
11:    {  aM = rhs.aM;
12:       cout << "A object copy created, aM = " << aM << endl; }
13:    A& operator=( const A& rhs ) // τελεστής εκχώρησης
14:    {  aM = rhs.aM;
15:       cout << "A assignment, aM = " << aM << endl;
16:       return *this;  }
17:    int getM() const {  return aM;  }
18: }; // A
19:
20: class C
21: {
22:    int cM;
23: public:
24:    C( int cp = 0 )    // ερήμην δημιουργός
25:    {  cM = cp;
26:       cout << "C object created, cM = " << cM << endl;  }
27:    C( const C& rhs )  // δημιουργός αντιγραφής
28:    {  cM = rhs.cM;
29:      cout << "C object copy created, cM = " << cM << endl; }
30:    C& operator=( const C& rhs ) // τελεστής εκχώρησης
31:    {  cM = rhs.cM;
32:       cout << "C assignment, cM = " << cM << endl;
33:       return *this;  }
34:    int getM() const {  return cM;  }
35: }; // C
36:
37: class D
38: {
39:    A dA;
40:    C dC;
41: public:
42:    D( A ap=A(0), C cp=C(0) )  // ερήμην δημιουργός
43:    {  dA = A(ap.getM()+1);  dC = C(cp.getM()+1);
44:       cout << "D object created" << endl;  }
45:    D( const D& rhs )          // δημιουργός αντιγραφής
46:    {  dA = rhs.dA;  dC = rhs.dC;
47:       cout << "D object copy created" << endl;  }
```

[3] Ο κανόνας δεν είναι πλήρης. Θα επανέλθουμε...

```
48: }; // D
49:
50: int main()
51: {
52:    D d;
53: }
```

θα δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 0
C object created, cM = 0
A object created, aM = 1
A assignment, aM = 1
C object created, cM = 1
C assignment, cM = 1
D object created
```

Πριν δημιουργηθεί το αντικείμενο *d* (γρ. 52 στη `main`): Πρώτα (γρ. 42) δημιουργούνται τα αντικείμενα-παράμετροι *cp* και *ap* (με αυτήν τη σειρά: πρόσεξε ότι η λίστα ορισμάτων σαρώνεται από το τέλος προς την αρχή πρώτα το *cp* και μετά το *ap*) του δημιουργού και μας δίνουν τις δύο πρώτες γραμμές από τους ερήμην δημιουργούς (με αρχική τιμή) των *A* και *C*.

Μετά αρχίζει η δημιουργία του *d*:

- Πρώτα θα δημιουργηθούν τα μέλη *da* και *dc* από τους ερήμην δημιουργούς των κλάσεων *A* και *C* (μηνύματα: «`A object created`», «`C object created`»), με τη σειρά που είναι δηλωμένα (γρ. 39, 40).
- Στη συνέχεια θα εκτελεσθεί το σώμα του δημιουργού της *D*: Καλείται ο δημιουργός (με αρχική τιμή) της *A* για να δημιουργήσει ένα αντικείμενο "`A(ap.getM()+1)`" (γρ. 43, μήνυμα: «`A object created, aM = 1`») που το εκχωρεί στο *da* («`A assignment, aM = 1`»). Παρόμοια συμβαίνουν και για το *dc* (μηνύματα: «`C object created, cM = 1`», «`C assignment, cM = 1`»). Τελικώς, θα μας δοθεί το μήνυμα: «`D object created`».

Αλλάζουμε τον δημιουργό της *D* (γρ. 42-44) ως εξής:

```
42:    D( A ap=A(0), C cp=C(0) )  // ερήμην δημιουργός
43:    : dA( A(ap.getM()+1) ),  dC( C(cp.getM()+1) )
44:    {  cout << "D object created" << endl;  }
```

δηλαδή: δίνουμε τιμές στα μέλη *da* και *dc* με **λίστα εκκίνησης** (initialization list) και όχι με τις εντολές εκχώρησης της γρ. 43. Τώρα η εκτέλεση θα δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 1
C object created, cM = 1
D object created
```

Ποια είναι η διαφορά από πριν; Τώρα τα δύο μέλη του *d* δημιουργούνται από τους δημιουργούς αντιγραφής με τις τιμές που πρέπει να έχουν. Έτσι *δεν χρειάζεται να γίνουν οι δύο εκχωρήσεις μέσα στο σώμα του δημιουργού* και ο δημιουργός γίνεται ταχύτερος.[4]

Παράδειγμα 2

Χρησιμοποιώντας λίστα εκκίνησης θα μπορούσαμε να γράψουμε τον δημιουργό της *Date* ως εξής:

```
   Date::Date( int yp, int mp, int dp )
      : year( yp ),  month( mp ),  day( dp )
   {
      if ( yp <= 0 )
         throw DateXptn( "Date", DateXptn::yearErr, yp );
      if ( mp <= 0 || 12 < mp )
```

[4] Στη συνέχεια θα δεις ότι δεν είναι μόνο θέμα ταχύτητας.

```
        throw DateXptn( "Date", DateXptn::monthErr, mp );
    if ( dp <= 0 || lastDay(year, month) < dp )
        throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
} // Date
```

Πρόσεξε ότι τώρα:

- Πρώτα δίνουμε τις τιμές στα μέλη –με τη δημιουργία τους– και μετά κάνουμε τους ελέγχους.
- *Οι έλεγχοι γίνονται πάντοτε στις τιμές των παραμέτρων και όχι των μελών*. (γιατί;)

Οι αρχικές τιμές που βάζουμε στη λίστα εκκίνησης είναι, γενικώς, σταθερές παραστάσεις των τυπικών παραμέτρων.

Παράδειγμα 3

Για την *complex* (§19.5) θα μπορούσαμε να γράψουμε:

```
complex::complex( double rp = 0.0, double ip = 0.0 )
    :  re( rp ), im( ip ) { };
```

Θα μπορούσαμε να γράψουμε έτσι και τον δημιουργό της *BString*; Όχι! Με τη "`bsData( rhs )`" *θα αντιγραφεί η τιμή του βέλους rhs στο βέλος bsData*.

Έτσι, οι δημιουργοί των κλάσεων εξαιρέσεων θα γραφούν ως εξής:

```
DateXptn( const char* mn, int ec, int ev1 = 0,
          int ev2 = 0, int ev3 = 0 )
    : errorCode( ec ),
      errVal1( ev1 ),  errVal2( ev2 ),  errVal3( ev3 )
{  strncpy( funcName, mn );  funcName[99] = '\0';  }
```

και

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

21.4 Εξαιρέσεις από τον Δημιουργό

Ο δημιουργός είναι μια συνάρτηση που πρέπει τελικώς να δημιουργήσει ένα αντικείμενο που συμμορφώνεται με την αναλλοίωτη.[5]

♦ ***Το (κάθε) αντικείμενο δημιουργείται όταν ολοκληρωθεί η εκτέλεση του δημιουργού.***

Απο εκεί και μετά μπορείς να το χειριστείς (και να το καταστρέψεις με τον καταστροφέα.) Η ζωή του αντικειμένου τελειώνει όταν αρχίσει η εκτέλεση του καταστροφέα του.

Αν δεν είναι δυνατό να έχουμε αντικείμενο που συμμορφώνεται με την αναλλοίωτη το καλύτερο που έχουμε να κάνουμε είναι να ρίξουμε εξαίρεση. Τι γίνεται όμως σε μια τέτοια περίπτωση; Έχουμε ένα «μισοδημιουργημένο» αντικείμενο; Όχι ή, εν πάση περιπτώσει, δεν έχουμε κάτι το οποίο μπορούμε να χειριστούμε.

Και τι γίνεται αν, πριν ριχτεί η εξαίρεση, έχουμε δεσμεύσει ήδη κάποιους πόρους του συστήματος; Θα πρέπει να τους απελευθερώσουμε.

Πριν δούμε σχετικά παραδείγματα ας πούμε τα καλά νέα: Αν κατά τη δημιουργία κάποιου μέλους έχουμε αποτυχία και ριχτεί εξαίρεση θα κληθούν οι καταστροφείς των μελών που ήδη δημιουργήθηκαν για να τα καταστρέψουν.

Πάμε τώρα στα δύσκολα με άλλο ένα παράδειγμα δημιουργού αντιγραφής: θα γράψουμε δημιουργό αντιγραφής για τη *Route*. Αντιγράφοντας αυτόν που γράψαμε για τη *BString* παίρνουμε:

---

[5] Η αναλλοίωτη της κλάσης είναι η *απαίτηση* για τον (κάθε) δημιουργό. Και ποια είναι η *προϋπόθεση*; Εξαρτάται από το είδος του δημιουργού.

```
Route::Route( const Route& rhs )
{
   try {  rAllStops = new RouteStop[rhs.rReserved];  }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rReserved = rhs.rReserved;
   for ( int k(0); k < rhs.rNoOfStops; ++k )
      rAllStops[k] = rhs.rAllStops[k];
   rNoOfStops = rhs.rNoOfStops;
   rCode = rhs.rCode;
   rFrom = rhs.rFrom;
   rTo = rhs.rTo;
   rInBetween = rhs.rInBetween;
} // Route::Route
```

Αυτός όμως δεν είναι σωστός. Κάθε **rAllStops[k]**, τύπου *RouteStop*, έχει ένα μέλος τύπου (*std*::)*string*. Η αντιγραφή του γίνεται όπως περίπου είδαμε στην επιφόρτωση του "**operator=**" για τη *BString* (§20.4)· δηλαδή για κάθε στοιχείο δεσμεύεται δυναμική μνήμη. Επομένως είναι δυνατόν να ριχτεί *bad_alloc* και όταν εκτελείται κάποια από αυτές τις αντιγραφές. Θα πρέπει λοιπόν, προβλέποντας μια τέτοια περίπτωση, να γράψουμε:

```
   try
   {
      for ( int k(0); k < rhs.rNoOfStops; ++k )
         rAllStops[k] = rhs.rAllStops[k];
   }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
```

Αλλά και αυτό είναι λάθος! Γιατί; Διότι έτσι θα έχουμε διαρροή μνήμης: η δυναμική μνήμη που πήραμε για τον **rAllStops** δεν έχει ανακυκλωθεί. Αυτό θα πρέπει να γίνει ως εξής:

```
   catch( bad_alloc )
   {  delete[] rAllStops;
      throw RouteXptn( "Route", RouteXptn::allocFailed );  }
```

και τελικώς θα έχουμε:

```
Route::Route( const Route& rhs )
{
   try {  rAllStops = new RouteStop[rhs.rReserved];  }
   catch( bad_alloc )
   {  throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rReserved = rhs.rReserved;
   try
   {
      for ( int k(0); k < rhs.rNoOfStops; ++k )
         rAllStops[k] = rhs.rAllStops[k];
      rFrom = rhs.rFrom;
      rTo = rhs.rTo;
      rInBetween = rhs.rInBetween;
   }
   catch( bad_alloc )
   {  delete[] rAllStops;
      throw RouteXptn( "Route", RouteXptn::allocFailed );  }
   rNoOfStops = rhs.rNoOfStops;
   rCode = rhs.rCode;
} // Route::Route
```

Πρόσεξε ότι μέσα στην περιοχή της (δεύτερης) **try** βάλαμε και τις αντιγραφές των *rFrom*, *rTo* και *rInBetween* αφού και αυτές μπορεί να δώσουν το ίδιο πρόβλημα. Αυτό θα πρέπει να το πάρουμε υπόψη μας και στις *setFrom*, *setTo* και *setInBetween*. Να τις ξαναγράψεις σωστά!

Γενικώς λοιπόν:

- ***Αν ρίξεις εξαίρεση από κάποιον δημιουργό φρόντισε πρώτα να αποδεσμεύσεις τους πόρους του συστήματος που έχεις δεσμεύσει για το αντικείμενο που δημιουργείς.***

Και αν η εξαίρεση ριχτεί από κάποια συνάρτηση που καλεί ο δημιουργός; Φροντίζουμε:

- να πιάσουμε την εξαίρεση στον δημιουργό,
- να αποδεσμεύσουμε τους πόρους που έχουμε δεσμεύσει και μετά
- ξαναρίχνουμε την εξαίρεση.

Αν θέλεις να λύσεις –τώρα ή αργότερα– ένα πρόβλημα τέτοιου είδους σκέψου την εξής περίπτωση: Στη *RouteStop* δηλώνουμε:

```
BString sName;  // όνομα στάσης
```

και έχουμε την εξής αλλαγή: Αν κάτι δεν πάει καλά σε κάποια από τις εκχωρήσεις "`rAllStops[k] = rhs.rAllStops[k]`" η εξαίρεση που θα πάρεις (δες την παρατήρηση που ακολουθεί) δεν θα είναι *bad_alloc* αλλά *BStringXptn*(με κωδικό *allocFailed*). Πώς θα τη χειριστείς;

Παρατήρησεις: ▶

1. Είπαμε ότι για τη *RouteStop* δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής. Όπως είδαμε όμως, *αυτό δεν σημαίνει ότι όταν αντιγράφουμε (με εκχώρηση) αντικείμενα αυτής της κλάσης δεν μπορεί, κατ' αρχήν, να ριχτεί εξαίρεση*. Προφανώς το ίδιο ισχύει και για αντιγραφή με τον δημιουργό αντιγραφής.
2. Τα πράγματα με την πρώτη **try** είναι πιο πολύπλοκα: Τι γίνεται αν μας δοθεί μνήμη για να δημιουργηθούν *m* (< *rhs*.*rReserved*) αντικείμενα και μετά εμφανισθεί το πρόβλημα; Έχουμε διαρροή μνήμης που δεσμεύτηκε για τα αντικείμενα αυτά! Η C++ έχει τρόπο (ριζικής) αντιμετώπισης όλων αυτών των προβλημάτων και θα τον μάθουμε αργότερα.◀

21.5 Ο Καταστροφέας

Ας ξεκινήσουμε με το πρόβλημα: έστω ότι έχουμε μια ομάδα μέσα στην οποία δηλώνουμε ένα αντικείμενο:

```
{
    BString a;
// . . .
} // T: εδώ πρέπει να καταστραφεί το a.
```

Το *a* είναι τοπικό στην ομάδα και θα πρέπει να καταστραφεί όταν η εκτέλεση φθάσει στο σημείο *T*, στο τέλος της ομάδας. Η C++ μας εγγυάται ότι θα επιστρέψει στη στοίβα τη μνήμη που κρατάμε για τα μέλη *bsData*, *bsLen*, *bsReserved* αλλά η δυναμική μνήμη που πήραμε από τον σωρό (και τη δείχνει το *bsData*) θα μείνει δεσμευμένη (και άχρηστη αφού το βέλος προς αυτήν ήταν το *bsData*). Όπως είδαμε, η C++ μας επιτρέπει να γράψουμε μια ειδική συνάρτηση-μέλος, που λέγεται **καταστροφέας** (destructor) που θα τον καλέσει όταν καταστρέφεται ένα αντικείμενο. Με τη συνάρτηση αυτή θα πρέπει να φροντίσουμε να επιστρέφεται η δυναμική μνήμη στον σωρό.

Για τη *BString* χρειάζεται να ορίσουμε *εμείς* καταστροφέα:

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

αλλά για τη *Date* δεν χρειάζεται. Για αυτήν, ο μεταγλωττιστής θα ορίσει αυτομάτως έναν *εννοούμενο* καταστροφέα· μπορείς να τον σκέφτεσαι κάπως έτσι:

```
   ~Date() { };
```

και είναι δεκτός. Γενικώς:

- ***Πρέπει να γράφουμε καταστροφέα για μια κλάση που τα αντικείμενά της δεσμεύουν πόρους του συστήματος, ώστε με την καταστροφή του αντικειμένου να αποδεσμεύονται οι πόροι.***

Αν έχουμε μια κλάση *K*:

- μπορούμε να γράψουμε *έναν καταστροφέα το πολύ* (ενώ μπορούμε να έχουμε περισσότερους από έναν δημιουργούς),
- ο καταστροφέας έχει όνομα "**~K**",
- ο καταστροφέας *δεν έχει παραμέτρους*.

Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα

```
~K() { };
```

στις κλάσεις που δεν χρειάζονται. Αργότερα θα δούμε ότι σε μερικές περιπτώσεις αυτό είναι απαραίτητο.

Με ποιες προδιαγραφές γράφουμε τον καταστροφέα;

Προϋπόθεση: Η αναλλοίωτη της κλάσης· το αντικείμενο που καταστρέφεται –όπως και κάθε αντικείμενο– συμμορφώνεται με την αναλλοίωτη της κλάσης. Για παράδειγμα, για να εκτελεσθεί η "**delete[] bsData**" η τιμή του *bsData* θα πρέπει να είναι είτε 0 (μηδέν) είτε μια διεύθυνση στη δυναμική μνήμη από όπου ξεκινάει η αποθήκευση πίνακα. Η αναλλοίωτη της *BString* –και πιο συγκεκριμένα η "0 < *bsReserved*"– μας εγγυάται την ύπαρξη του δυναμικού πίνακα.

Απαίτηση: Οι πόροι –που ήταν δεσμευμένοι από το αντικείμενο που καταστράφηκε– έχουν αποδεσμευθεί.

Τώρα θα κάνουμε ένα πείραμα –με ένα προγραμματάκι– για να δούμε πότε καλείται ένας καταστροφέας. Αλλάζουμε τον καταστροφέα της *BString* σε:

```
BString::~BString()
{
cout << "destructing BString " << c_str() << endl;
   delete[] bsData;
} // BString::~BString
```

και δοκιμάζουμε το

```
#include <iostream>
#include "BString.cpp"
using namespace std;

struct S
{
   BString sName;
   float   sDist;
}; // S

int main()
{
   {
      BString a( "aaa" );
// . . .
      cout << "going to destroy a" << endl;
   } // T: εδώ πρέπει να καταστραφεί το a.

   BString* b( new BString("bbb") );
// . . .
   cout << "going to delete b" << endl;
   delete b;

   BString* c( new BString[2] );
   c[0] = "c0c0";  c[1] = "c1c1";
// . . .
   cout << "going to delete c" << endl;
   delete[] c;
```

```
   {
      S s;
      s.sName = "sssss";  s.sDist = 7.33;
// . . .
      cout << "going to destroy s" << endl;
   } // T: εδώ πρέπει να καταστραφεί το s.

   S* s( new S );
   s->sName = "psps";  s->sDist = 7.33;
// . . .
   cout << "going to delete s" << endl;
   delete s;
}
```

Αποτέλεσμα:

```
going to destroy a
destructing BString aaa
going to delete b
destructing BString bbb
going to delete c
destructing BString c1c1
destructing BString c0c0
going to destroy s
destructing BString sssss
going to delete s
destructing BString psps
```

Τι βλέπουμε εδώ;

- Κατ' αρχάς δίνουμε το παράδειγμα με το οποίο ξεκινάει αυτή η παράγραφος: «*Το a είναι τοπικό στην ομάδα και θα πρέπει να καταστραφεί όταν η εκτέλεση φθάσει στο σημείο T, στο τέλος της ομάδας.*» Εδώ το βλέπεις.
- Στα δύο επόμενα βλέπεις το εξής: αφού για να δημιουργηθούν δυναμικές μεταβλητές ή πίνακες καλείται ο δημιουργός, για να ανακυκλωθούν καλείται ο καταστροφέας.
- Τέλος, στα δύο τελευταία παραδείγματα βλέπεις το εξής: Ο καταστροφέας των αντικειμενων μιας κλάσης –εδώ ο εννοούμενος της κλάσης *S*– καλεί τους καταστροφείς των μελών των αντικειμένων.

Πώς γίνεται η καταστροφή του αντικειμένου;

♦ ***Η καταστροφή ενός αντικειμένου γίνεται με την αντίστροφη σειρά από αυτήν που γίνεται η δημιουργία του.***

Με αυτά που ξέρουμε μέχρι τώρα:

- Εκτελείται το σώμα του καταστροφέα.
- Καταστρέφονται τα μέλη με αντίστροφη σειρά από αυτήν που δηλώνονται.

Μπορείς να δεις ένα παράδειγμα πολύ εύκολα. Εφοδίασε τις κλάσεις του Παραδ. 1 της §21.3 με τους καταστροφείς:

```
   ~A()
   {  cout << "destructing an A object aM = " << aM << endl; }

   ~C()
   {  cout << "destructing a C object aM = " << cM << endl; }

   ~D()
   {  cout << "destructing a D object" << endl; }
```

και η εκτέλεση του προγράμματος θα μας δώσει:

```
C object created, cM = 0
A object created, aM = 0
A object created, aM = 1
C object created, cM = 1
D object created
```

```
destructing an A object aM = 0
destructing a C object aM = 0
destructing a D object
destructing a C object aM = 1
destructing an A object aM = 1
```

Οι δύο πρώτες καταστροφές προέρχονται από τις καταστροφές των "`A(0)`" και "`C(0)`" και δεν έχουν σχέση με την καταστροφή του *d*· αυτή ξεκινάει με την εκτέλεση του σώματος του καταστροφέα που μας δίνει το μήνυμα "`destructing a D object`". Στη συνέχεια βλέπεις ότι καταστρέφονται πρώτα το *dc* και μετά το *da*.

21.5.1 Ο Καταστροφέας *δεν* Ρίχνει Εξαιρέσεις

Ένα θέμα που χρειάζεται προσοχή είναι οι εξαιρέσεις του καταστροφέα:

- ***Ένας καταστροφέας όχι μόνο δεν ρίχνει εξαιρέσεις αλλά δεν επιτρέπει να βγαίνουν από αυτόν εξαιρέσεις από συναρτήσεις που καλεί.***

Γιατί; Σκέψου την εξής περίπτωση: Μέσα σε μια συνάρτηση ρίχνεται μια εξαίρεση και αρχίζει το «ξετύλιγμα της στοίβας». Πρώτο βήμα: σταματάει η εκτέλεση της συνάρτησης και καταστρέφονται όλα τα τοπικά αντικείμενά της. Τι θα γίνει αν κατά την καταστροφή κάποιου από αυτά τα αντικείμενα πέσει μέσα στον καταστροφέα μια εξαίρεση; Σταματάει η εκτέλεση του καταστροφέα, δηλαδή η καταστροφή του αντικειμένου. Όποιο σενάριο διαχείρισης εξαιρέσεων και να έχεις στήσει καταρρέει (μαζί και το πρόγραμμα).

Τώρα θα πεις: τι εξαίρεση από τον καταστροφέα, από μισή γραμμή συνάρτηση;! Ε, έτσι είναι στη *BString*. Υπάρχουν και πιο πολύπλοκοι καταστροφείς· στη συνέχεια θα δούμε ένα παράδειγμα.

21.5.2 Καλούμε τον Καταστροφέα;

Σε όλα τα παραδείγματα που δώσαμε παραπάνω ο καταστροφέας καλείται αυτομάτως χωρίς εμείς να γράψουμε κάτι στο πρόγραμμά μας. Υπάρχει περίπτωση να χρειαστεί να τον καλέσουμε με δική μας εντολή όπου να βλέπουμε κάτι σαν "`~BString()`" ή "`~Date()`"; Σπανίως! Δες ένα παράδειγμα, αφού πρώτα κάνεις μια επανάληψη στην «τρίτη μορφή του "`new`"» (§16.5).

Αλλάζοντας λίγο το παράδειγμα που δίναμε εκεί, γράφουμε:

```
   char* buf( new char[100] );
   BString* bs( new (buf) BString );
// . . .
   delete[] buf;
```

Τα μέλη της `*`*bs* θα υλοποιηθούν «πάνω» στα στοιχεία του buf. Για παράδειγμα, τα μέλη *bsData*, *bsLen*, *bsReserved* θα μπορούσαν να υλοποιηθούν ξεκινώντας από buf`[0]`, buf`[4]`, buf`[8]` αντιστοίχως. Αλλά η (δυναμική) μνήμη που δείχνει το *bsData* θα παραχωρηθεί αλλού (όχι πάνω στον buf). Όταν εκτελεσθεί η "`delete[] buf`" θα ανακυκλωθούν όλα τα μέλη του `*`*bs*, μεταξύ αυτών και το βέλος *bs*`->`*bsData*, αλλά θα μας μείνει αυτό που δείχνει. Έχουμε δηλαδή διαρροή μνήμης. Πώς το αποφεύγουμε; Βάζοντας:

```
// . . .
   bs->~BString();
   delete[] buf;
```

Το ίδιο θα πρέπει να κάνουμε και αν θέλουμε να αλλάξουμε τη χρήση της buf:

```
   char* buf( new char[100] );
   BString* bs( new (buf) BString );
// . . .
   bs->~BString();
   double* d( new (buf) double );
// . . .
```

```
   delete[] buf;
```

Παρατήρηση: ▶
Μετά την κλήση "`bs->~BString()`"ο δυναμικός πίνακας που δείχνει το βέλος *bs->bsData* δεν υπάρχει πια· έχει ανακυκλωθεί. Το *bs->bsReserved* έχει θετική τιμή αλλά είναι ψεύτικη.◀

21.6 Ο Τελεστής Εκχώρησης

Στο προηγούμενο κεφάλαιο (§20.4) είδαμε ότι το πρόβλημα που μας υποχρέωνε να γράψουμε δημιουργό αντιγραφής για τη *BString* μας υποχρέωνε να επιφορτώσουμε και τον τελεστή εκχώρησης που τον ορίσαμε ως εξής:

```
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=
```

Κάναμε μάλιστα την επισήμανση: «*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.*» Συνεπώς, αν κατά την εκτέλεση της "`a = b`" «μείνουμε» από μνήμη θα ριχτεί εξαίρεση αλλά δεν θα καταστραφεί η παλιά τιμή της *a*. Όπως καταλαβαίνεις, έτσι αφήνουμε αρκετές επιλογές –στο πρόγραμμα που χρησιμοποιεί την κλάση– για τη διαχείριση της κατάστασης.

Θα τηρούμε λοιπόν γενικώς τον εξής προγραμματιστικό κανόνα:

- ***Κάθε μέθοδος που αποπειράται να αλλάξει την τιμή ενός αντικειμένου (εκχώρηση, ανάγνωση από αρχείο) αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.***

Λέμε ότι η μέθοδος αυτή έχει **ασφάλεια προς τις εξαιρέσεις** (exception safety) επιπέδου **ισχυρής εγγύησης** (strong guarantee). Αργότερα θα δούμε την ασφάλεια προς τις εξαιρέσεις πιο εκτεταμένα.

21.6.1 Η Ασφαλής *swap()*

Εδώ αξίζει να αναφέρουμε μια κομψή μορφή του τελεστή εκχώρησης που προτείνεται από τον (Stepanov[6], 2007) και έχει ασφάλεια ισχυρής εγγύησης. Αυτή η μορφή βασίζεται στον δημιουργό αντιγραφής και σε μια ασφαλή μέθοδο *swap()*.

[6] A.A. Stepanof, ρώσος προγραμματιστής (και όχι μόνον), «πατέρας» της STL.

Ας ξεκινήσουμε με τη *BString*. Αν θέλαμε να γράψουμε μια μέθοδο *swap()*, με αυτά που ξέρουμε μέχρι τώρα, θα εξειδικεύαμε («με το χέρι») το περίγραμμα της §14.7.1:

```
void BString::swap( BString& rhs )
{
   BString s( *this );
   *this = rhs;  rhs = s;
} // BString::swap
```

Εδώ όμως χρησιμοποιούμε τον δημιουργό αντιγραφής και τον τελεστή εκχώρησης που μπορεί να ρίξουν εξαιρέσεις. Επομένως δεν είναι ασφαλής. Το πιο βασικό πρόβλημα είναι ότι θέλουμε να τη χρησιμοποιήσουμε *για να ορίσουμε τον τελεστή εκχώρησης* και αυτή η μορφή τον χρησιμοποιεί!

Μπορούμε να γράψουμε μια ασφαλή *swap()* αν κάνουμε τις ανταλλαγές τιμών μέλος-προς-μέλος, δηλαδή:

```
void BString::swap( BString& other )
{
   char* pSave( bsData );
   bsData = other.bsData;  other.bsData = pSave;

   size_t save( bsLen );
   bsLen = other.bsLen;  other.bsLen = save;

   save = bsReserved;
   bsReserved = other.bsReserved;  other.bsReserved = save;
} // BString::swap
```

Πρόσεξε τα εξής:

- Στην πρώτη αντιμετάθεση έχουμε ανταλλαγή τιμών των *βελών bsData* και *rhs.bsData* και όχι των ορμαθών που δείχνουν.
- Η ασφάλεια της μεθόδου στηρίζεται στο ότι ανταλλάσσονται απλές τιμές τύπου `char*` ή `size_t`. Δεν καλεί δημιουργό ή τον τελεστή εκχώρησης της *BString*.[7]

Τώρα, μπορούμε να γράψουμε:[8]

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      BString tmp( rhs );
      swap( tmp );
   }
   return *this;
} // BString::operator=
```

Αν ρίξει εξαίρεση ο δημιουργός αντιγραφής, η εκτέλεση της "`BString tmp(rhs)`" δεν θα ολοκληρωθεί, η *swap()* δεν θα κληθεί και το `*this` δεν θα πειραχτεί. Αν το *tmp* δημιουργηθεί κανονικώς τότε όλα θα πάνε καλώς αφού η *swap()* αποκλείεται να ρίξει εξαίρεση.

Σύγκρινε αυτήν τη μορφή με αυτήν της προηγούμενης παραγράφου: στην πρώτη μορφή ήταν υποχρέωση δική μας να δεσμεύσουμε πόρους (δυναμική μνήμη) και να τη απελευθερώσουμε. Αυτά πρέπει να γίνουν με προσοχή, όπως ήδη είδαμε. Στη δεύτερη μορφή:

- Η δέσμευση πόρων γίνεται με τη δημιουργία ενός αντικειμένου (*tmp*).
- Η απελευθέρωση των πόρων γίνεται με την καταστροφή του αντικειμένου.

---

[7] Λέμε ότι η *swap()* είναι ασφαλής προς τις εξαιρέσεις σε επίπεδο **εγγύησης μη-ρίψης** (no-throw guarantee).

[8] Αυτή η μορφή επιφόρτωσης του "`operator=`" θα δεις να αναφέρεται και ως *κανονική μορφή* (canonical form).

Χρησιμοποιούμε δηλαδή την τεχνική RAII με πολύ πιο φυσικό τρόπο από αυτόν που είδαμε στο παράδειγμα της §16.7.1.

Θα πρέπει να επισημάνουμε ακόμη την τυποποιημένη μορφή· αν θελήσεις να τον γράψεις για τη *Route* θα έχεις:

```
Route& Route::operator=( const Route& rhs )
{
   if ( &rhs != this )
   {
      Route tmp( rhs );
      swap( tmp );
   }
   return *this;
} // Route::operator=
```

Βέβαια, θα πρέπει να γράψουμε τη *swap()*· αυτό σου το αφήνουμε ως άσκηση.

Θα δηλώσουμε τη *swap()* ως "`private`" ή "`public`"; Δεν έχουμε λόγο να τη δηλώσουμε "`private`"· πέρα από τη χρήση της στις άλλες μεθόδους, την παρέχουμε ως ένα επιπλέον εργαλείο προς τον προγραμματιστή-χρήστη της κλάσης. Θυμίσου άλλωστε ότι η *string::swap()* είναι "`public`".

Παρατηρήσεις: ▶

1. Αν χρησιμοποιήσεις το περίγραμμα `std::swap()` μπορείς να γράψεις πολύ πιο απλά:

```
void BString::swap( BString& other )
{
   std::swap( bsData, other.bsData );
   std::swap( bsLen, other.bsLen );
   std::swap( bsReserved, other.bsReserved );
} // BString::swap
```

Στη συνέχεια θα χρησιμοποιούμε αυτήν τη μορφή.

2. Όποιος διάβασε με προσοχή τα προηγούμενα μπορεί να έχει σκεφτεί ήδη μια βελτίωση. Αν αντί για "`const BString& rhs`" βάλουμε παράμετρο τιμής, δηλαδή "`BString rhs`" όταν εκτελείται η "`a = b`" θα κληθεί ο δημιουργός αντιγραφής για να δημιουργήσει ένα τοπικό αντίγραφο της *b* στην *rhs* που θα καταστραφεί όταν τελειώσει η εκτέλεση της συνάρτησης. Έτσι, η *tmp* μας είναι άχρηστη!

```
BString& BString::operator=( BString rhs )
{
   if ( &rhs != this )
   {
      swap( rhs );
   }
   return *this;
} // BString::operator=
```

Στην περίπτωση αυτήν *θα γίνεται πάντοτε αντιγραφή στο τοπικό αντικείμενο* ενώ στην πρηγούμενη μορφή δεν γίνεται στην περίπτωση αυτοεκχώρησης. Σωστό! Αλλά πόσες αυτόεκχωρήσεις θα ζητηθούν σε κάποιο πρόγραμμα;

3. Αν υπάρξει κάποιο πρόβλημα, κατά τη χρήση αυτού του τελεστή, η εξαίρεση που θα ριχτεί θα δείχνει προέλευση τον δημιουργό (αντιγραφής). Αυτό φυσικά είναι παραπλανητικό· ας το διορθώσουμε:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
```

```
} // BString::operator=
```

Δηλαδή: πιάνουμε την όποια εξαίρεση *BStringXptn*, διορθώνουμε το *funcName* και την ξαναρίχνουμε.◀

Η χρήση της ασφαλούς *swap()* μπορεί να μας δώσει ισχυρή εγγύηση ασφάλειας προς τις εξαιρέσεις όχι μόνον για τον τελεστή εκχώρησης αλλά και για άλλες μεθόδους που αλλάζουν την τιμή του αντικειμένου, π.χ. μεθόδους ανάγνωσης από αρχείο. Θα δώσουμε τώρα ένα τέτοιο παράδειγμα: Θα λύσουμε μερικώς την Άσκ. 20-5. Θα γράψουμε μια μέθοδο

```
void RouteStop::readFromText( istream& tin )
```

που θα διαβάζει τιμή αντικειμένου *RouteStop* από αρχείο και θα είναι ασφαλής προς τις εξαιρέσεις: το αντικείμενο-ιδιοκτήτης της θα κρατάει την τιμή που έχει αν ριχτεί εξαίρεση. Θα χρησιμοποιήσουμε τη:

```
void RouteStop::swap( RouteStop& other )
{
   sName.swap( other.sName );
   std::swap( sDist, other.sDist );
   std::swap( sFare, other.sFare );
} // RouteStop::swap
```

Να σημειώσουμε ότι η `string::swap()` είναι ασφαλής.

Το *tin* είναι ρεύμα από ένα αρχείο text που είναι ήδη ανοικτό για διάβασμα. Σε κάθε γραμμή του αρχείου αναμένουμε τα στοιχεία μιας στάσης:

```
  0: void RouteStop::readFromText( istream& tin )
  1: {
  2:    try
  3:    {
  4:       string line;
  5:       getline( tin, line, '\n' );
  6:       if ( tin.fail() )
  7:          throw RouteStopXptn( "readFromText",
  8:                               RouteStopXptn::cannotRead );
  9:       if ( line.empty() )
 10:          throw RouteStopXptn( "readFromText",
 11:                               RouteStopXptn::lineEmpty );
 12:       size_t t1Ndx( line.find('\t', 0) );
 13:       if ( t1Ndx == string::npos )
 14:          throw RouteStopXptn( "readFromText",
 15:                               RouteStopXptn::incomplete );
 16:       size_t t2Ndx( line.find('\t', t1Ndx+1) );
 17:       if ( t1Ndx == string::npos )
 18:          throw RouteStopXptn( "readFromText",
 19:                               RouteStopXptn::incomplete );
 20:       RouteStop tmp;
 21:       if ( t1Ndx == 0 )
 22:          throw RouteStopXptn( "readFromText",
 23:                               RouteStopXptn::noName );
 24:       tmp.sName = line.substr( 0, t1Ndx );
 25:       string buf( line.substr(t1Ndx+1, t2Ndx-t1Ndx-1) );
 26:       tmp.sDist = atof( buf.c_str() );
 27:       if ( tmp.sDist < 0 )
 28:          throw RouteStopXptn( "readFromText",
 29:                               RouteStopXptn::negDist,
 30:                               tmp.sDist );
 31:       buf = line.substr( t2Ndx+1 );
 32:       tmp.sFare = atof( buf.c_str() );
 33:       if ( tmp.sFare < 0 )
 34:          throw RouteStopXptn( "readFromText",
 35:                               RouteStopXptn::negFare,
 36:                               tmp.sFare );
 37:       this->swap( tmp );
 38:    }
 39:    catch( bad_alloc )
 40:    {  throw RouteStopXptn( "readFromText",
```

```
41:                                RouteStopXptn::allocFailed );  }
42: } // RouteStop::readFromText
```

Μετά τη δήλωση της γρ. 4, διαβάζουμε με τη *getline()* (γρ. 5). Αν η *getline()* εκτελέστηκε κανονικώς θα προχωρήσουμε, αλλιώς... εξαίρεση (γρ. 6-8).

Αν διαβάσαμε άδεια γραμμή και πάλι ρίχνουμε εξαίρεση (γρ. 9-11).

Αν δεν είναι άδεια *θα πρέπει να έχει δύο φορές τον* `'\t'`:

- Χρησιμοποιώντας τη *find()* της *string* επιδιώκουμε να αποθηκεύσουμε τη θέση του πρώτου στην *t1Ndx* (γρ. 12).
- Και πάλι με τη *find()* αλλά ξεκινώντας από τη θέση *t1Ndx*+1 επιδιώκουμε να αποθηκεύσουμε τη θέση του δεύτερου στην *t2Ndx* (γρ. 16).

Αποτυχία οποιασδήποτε από τις δύο *find()* οδηγεί στη ρίψη εξαίρεσης (γρ. 13-15, 17-19).

Αφού ο πρώτος `'\t'` βρίσκεται στη θέση *t1Ndx* το όνομα της στάσης βρίσκεται στις θέσεις 0 `..` *t1Ndx*–1 (εκτός από την περίπτωση που *t1Ndx* `==` 0). Αφού ο δεύτερος `'\t'` βρίσκεται στη θέση *t2Ndx* στις θέσεις *t1Ndx*+1 `..` *t2Ndx*–1 υπαρχει η απόσταση και από *t2Ndx*+1 μέχρι το τέλος της γραμμής υπαρχει το κόμιστρο. Όλα αυτά θα αποθηκευτούν σε μια βοηθητική μεταβλητή (γρ. 20). Παραλλήλως θα γίνονται και οι σχετικοί έλεγχοι (γρ. 21-36).

Αν δεν χρειαστεί να ρίξουμε εξαίρεση, στο τέλος του δρόμου δίνουμε:

```
   this->swap( tmp );
```

ή απλώς: `swap( tmp )`.

Όλα αυτά γίνονται υπό έλεγχο για *bad_alloc* που μπορεί να έλθει –τουλάχιστον κατ' αρχήν– από τη διαχείριση δεδομένων τύπου *string*.

Παρατηρήσεις: ▶

1. Σχετικώς με τη *RouteStopXptn*`::`*lineEmpty*: Είναι τόσο τρομερό πρόβλημα που πρέπει να ρίξουμε εξαίρεση; Λοιπόν, πρόσεξε: Η μέθοδος έχει υποχρέωση να γνωστοποιήσει το γεγονός «*με έβαλες να διαβάσω μια γραμμή με στοιχεία στάσης και η γραμμή ήταν άδεια!*» Το «*ε, καλά, δεν χάθηκε ο κόσμος*» μπορεί να το πει το πρόγραμμα που καλεί τη *readFromText()*. Στην περίπτωσή μας αυτό μπορεί να γίνει αν αλλάξουμε τον τρόπο ανάγνωσης ως εξής:

```
      do {
         try {
             RouteStop oneStop;
             oneStop.readFromText( tin );
             oneRoute.insert1RouteStop( oneStop );
         }
         catch( RouteStopXptn& x )
         {
            if ( x.errCode != RouteStopXptn::lineEmpty &&
                 x.errCode != RouteStopXptn::cannotRead )
               throw;
         }
      } while ( !tin.eof() );
```

2. Γιατί να βάλουμε "`this->swap(tmp)`" και όχι "`*this = tmp`"; Διότι η δεύτερη θα εκτελέσει, μεταξύ άλλων, την "`sName = tmp.sName`" που, κατ' αρχήν, δεν είναι ασφαλής. Η *swap()* μας καλύπτει από την εξής περίπτωση: έχουμε περάσει επιτυχώς όλους τους ελέγχους και παίρνουμε εξαίρεση *bad_alloc* όταν προσπαθούμε να αντιγράψουμε το όνομα της στάσης.◀

21.7 * Προσωρινά Αντικείμενα

Στην §19.1 είδαμε την εντολή:

```
   d1 = Date( 2008, 7, 9 );
```

ενώ στην §20.4.2 είπαμε ότι η "`s2 = "what\'s this\?"`" είναι δεκτή διότι ο μεταγλωττιστής την καταλαβαίνει σαν:

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

Αυτά είναι δύο παραδείγματα όπου καλείται δημιουργός με αρχική τιμή να δημιουργήσει ένα **προσωρινό** (temporary) αντικείμενο *χωρίς όνομα*. Και στις δύο περιπτώσεις η τιμή του προσωρινού αντικειμένου εκχωρείται στο αντικείμενο που βρίσκεται στο αριστερό μέρος της εκχώρησης. Άλλη συνηθισμένη περίπτωση είναι η δημιουργία προσωρινού αντικειμενου σε κάποιο όρισμα όταν καλείται μια συνάρτηση.

Πόσο ζει ένα προσωρινό αντικείμενο; Μέχρι το τέλος του υπολογισμού της παράστασης μέσα στην οποία εμφανίζεται. Στα παραδείγματά μας, το κάθε ένα από τα προσωρινά αντικείμενα ζει μέχρι το τέλος της εκτέλεσης της αντίστοιχης εκχώρησης.

21.8 Ο «Κανόνας των Τριών»

Από όσα είπαμε μέχρι τώρα θα πρέπει να σου φαίνεται αυτονόητος ο «Κανόνας των Τριών» (rule of three ή Law of The Big Three) που λέει ότι

- ***Αν μια κλάση χρειάζεται κάποιο από τα: Δημιουργό Αντιγραφής, Τελεστή Εκχώρησης, Καταστροφέα, πιθανότατα χρειάζεται και τα άλλα δύο.***

Η *BString* και η *Route* είναι χαρακτηριστικά παραδείγματα εφαρμογής του κανόνα.

Με την ευκαιρία, να επανέλθουμε και στο ερώτημα: *πότε απαιτείται να ορίσουμε εμείς αυτά τα τρία εργαλεία;* Η απάντήση που δώσαμε είναι:

- ***Απαιτείται να ορίσουμε Δημιουργό Αντιγραφής, Τελεστή Εκχώρησης και Καταστροφέα, όταν τα αντικείμενα της κλάσης διαχειρίζονται πόρους του συστήματος που παίρνουν δυναμικώς.***

21.9 Μια Παρένθεση για τη *renew()*

Εδώ θα πρέπει να κάνουμε μια παρένθεση για να διορθώσουμε τη *renew()* που είδαμε στην §16.12 όπως είχαμε υποσχεθεί στη δεύτερη παρατήρηση της παραγράφου. Να θυμίσουμε ότι είχαμε:

```
try
{
   T* temp( new T[nf] );
   for ( int k(0); k < ni; ++k )  temp[k] = p[k];
   delete[] p;   p = temp;
}
catch( bad_alloc& )
{
   throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed );
}
```

και στην παρατήρηση λέγαμε «*Η bad_alloc μπορεί να προέλθει όχι μόνο από τη "`new T[nf]`" αλλά και από τις αντιγραφές "`temp[k] = p[k]`". Αυτό μπορεί να γίνει αν ο T έχει αντικείμενα που χρησιμοποιούν δυναμική μνήμη.*»

Ας ξαναγράψουμε τα παραπάνω πιο προσεκτικά:

```
T* temp;
try
{  temp = new T[nf];  }
catch( bad_alloc& )
{  throw MyTmpltLibXptn( "renew",
                         MyTmpltLibXptn::allocFailed ); }
try
{
   for ( int k(0); k < ni; ++k )  temp[k] = p[k];
```

```
   }
   catch( bad_alloc& )
   {  delete[] temp;
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::allocFailed ); }
   delete[] p;   p = temp;
```

Δηλαδή, αν πάρουμε μνήμη για τα *nf* αντικείμενα τύπου *T* που δείχνει το *temp* αλλά στη συνέχεια αποτύχουμε στην προσπάθεια να πάρουμε μνήμη για να κάνουμε τις αντιγραφές, φροντίζουμε –πριν ρίξουμε την εξαίρεση– να ανακυκλώσουμε τον πίνακα που δείχνει η *temp*.

Ας πούμε τώρα ότι θέλουμε να μεγαλώσουμε έναν δυναμικό πίνακα με στοιχεία κλάσης *BString*:

- Όπως γράψαμε τον τελεστή εκχώρησης της κλάσης (§20.4), όταν βρει πρόβλημα με τη δυναμική μνήμη, ρίχνει εξαίρεση κλάσης *BStringXptn* με κωδικό σφάλματος *BStringXptn* `::`*allocFailed*.
- Παρόμοια εξαίρεση μπορεί να ριχτεί και από τον ερήμην δημιουργό που καλείται για να δημιουργήσει όλα τα στοιχεία του πίνακα που δείχνει το *temp*.

Τι πρέπει να κάνουμε; Και στις δύο **try** θα πρέπει να βάλουμε και δεύτερη **catch** που θα πιάνει τις *BStringXptn*.

Στην πρώτη περίπτωση θα έχουμε:

```
   catch( bad_alloc& )
   {  throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed ); }
   catch( BStringXptn& x )
   {
      if ( x.errorCode == BStringXptn::allocFailed )
         throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed);
      else
         throw;
   }
```

και στη δεύτερη:

```
   catch( bad_alloc& )
   {  delete[] temp;
      throw MyTmpltLibXptn( "renew",
                            MyTmpltLibXptn::allocFailed ); }
   catch( BStringXptn& x )
   {
      delete[] temp;
      if ( x.errorCode == BStringXptn::allocFailed )
         throw MyTmpltLibXptn( "renew", MyTmpltLibXptn::allocFailed);
      else
         throw;
   }
```

Για να πάρουμε υπόψη μας όλα αυτά θα πρέπει να επιφορτώσουμε ένα δεύτερο περίγραμμα

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

στο οποίο θα ενσωματώσουμε τα παραπάνω αντικαθιστώντας το *BStringXptn* με *TXptn*. Σε κάθε στιγμιότυπο, στη θέση αυτής της παραμέτρου, θα πρέπει να βάζουμε κλάση εξαιρέσεων, σαν αυτές που γράφουμε εδώ, που να έχει κωδικό σφάλματος με όνομα *allocFailed*.

Τώρα πρόσεξε: *ειδικώς για τα παραδείγματα που δίνουμε σε αυτό το βιβλίο μπορούμε να απλουστεύσουμε τη renew()* αν πάρουμε υπόψη μας τα εξής:

- Αν δεν μπορέσουμε να πάρουμε δυναμική μνήμη για τον δυναμικό πίνακα που δείχνει η *temp* θα πάρουμε *bad_alloc*.

- Ο ερήμην δημιουργός δημιουργεί τα στοιχεία του δυναμικού πίνακα με τις ερήμην καθορισμένες τιμές. Δεν θα μπορέσει να δημιουργήσει κάποια από τα στοιχεία του δυναμικού πίνακα αν δεν μπορέσει να δεσμεύσει κάποιον πόρο του συστήματος. Στα παραδείγματά μας ο μόνος πόρος που διαχειριζόμαστε είναι η δυναμική μνήμη· άρα θα πάρουμε και πάλι *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.
- Ο αντιγραφικός τελεστής εκχώρησης αντιγράφει το *έγκυρο* αντικείμενο `p[k]` στο `temp[k]`. Η μόνη περίπτωση αποτυχίας της αντιγραφής έχει να κάνει με αποτυχία δέσμευσης πόρων. Για τα παραδείγματά μας αυτό σημαίνει αποτυχία δέσμευσης δυναμικής μνήμης, δηλαδή *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.

Μπορούμε λοιπόν να χρησιμοποιούμε της εξής μορφή:

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

όπου θεωρούμε ότι το μόνο πράγμα που μπορεί να «πάει στραβά» είναι η αποτυχία παραχώρησης δυναμικής μνήμης.

Αν χρησιμοποιήσεις τη *renew()* όταν τα αντικείμενα της *T* δεσμεύουν άλλους πόρους θα πρέπει να χρησιμοποιήσεις την πιο πολύπλοκη μορφή.[9]

Να τονίσουμε ότι όλες οι παραπάνω μορφές της *renew()* είναι ασφαλείς ως προς τις εξαιρέσεις με την εξής έννοια: αν ριχτεί εξαίρεση

- δεν έχουμε διαρροή μνήμης και
- δεν αλλάζει ο αρχικός πίνακας.

21.10 Αυτά που Μάθαμε στην Πράξη: ΑΝ ΔΕ {ΔΑ ΤΕ ΚΑ} GE SE

Στην §20.8 παραθέσαμε μερικά πράγματα που πρέπει να σκεφθούμε όταν γράφουμε μια κλάση. Τώρα μπορούμε να τα δούμε πιο συστηματικά.

Όπως θα πρέπει να έγινε φανερό, από όσα είπαμε μέχρι τώρα, για κάθε κλάση, πέρα από οποιεσδήποτε άλλες ειδικές απαιτήσεις, θα πρέπει να εξετάζουμε –και να υλοποιούμε όταν χρειάζεται– τα εξής:

- αναλλοίωτη (**ΑΝ**, class invariant),
- ερήμην δημιουργό (**ΔΕ**, default constructor),

[9] που θα πρέπει να την ολοκληρώσεις.

- δημιουργό αντιγραφής (**ΔΑ**, copy constructor),
- τελεστή εκχώρησης (**ΤΕ**, "**=**"),
- καταστροφέα αντικειμένων (**ΚΑ**, destructor),
- συναρτήσεις "*get*" (**GE**) για όλα τα χαρακτηριστικά,
- συναρτήσεις "*set*" (**SE**) για όλα τα χαρακτηριστικά.

Πάντοτε ή όταν μας χρειάζονται στην εφαρμογή-πελάτη της κλάσης; Είναι πολύ λίγες οι περιπτώσεις που η χρήση μιας κλάσης περιορίζεται σε μια μόνον εφαρμογή. Η εφαρμογή που θα χρησιμοποιήσει για πρώτη φορά μια κλάση μπορεί απλώς να σε οδηγήσει στην υλοποίηση επιπλέον μεθόδων ή/και στην αχρήστευση μερικών μεθόδων "*get*" ή/και "*set*". Αυτό το τελευταίο το είδαμε ήδη στην κλάση *Battery*: δεν γράψαμε μεν *getEnergy*, *setEnergy*, *setMaxEnergy*, *setVoltage* αλλά γράψαμε μεθόδους που –μαζί με τον δημιουργό– δίνουν μεγαλύτερη λειτουργικότητα.

Ας τα πάρουμε ένα-ένα:

Αναλλοίωτη (ΑΝ, class invariant): Η αναλλοίωτη είναι μέρος

- της απαίτησης κάθε δημιουργού (αλλιώς: ισχύει αμέσως μετά τη δημιουργία του κάθε αντικειμένου),
- της προϋπόθεσης του καταστροφέα,
- της προϋπόθεσης και της απαίτησης κάθε μεθόδου ενός αντικειμένου που έχει δηλωθεί "`public`".

Έτσι, μεταξύ άλλων είναι απαραίτητη για να γράψουμε σωστά τις μεθόδους *set* και δημιουργούς με αρχικές τιμές. Θα πρέπει να γράφεται με μαθηματική διατύπωση; Όχι κατ' ανάγκη· μπορεί να καταγραφεί και σε μορφή κειμένου. Αλλά, όσο ακριβέστερα κατάγραφεί τόσο καλύτερα χρησιμοποιείται.

Ερήμην Δημιουργός (ΔΕ, default constructor): Ο ερήμην δημιουργός καλείται για τη δημιουργία: στατικών ή δυναμικών μεταβλητών χωρίς αρχική τιμή, και στατικών ή δυναμικών πινάκων. Σχεδόν απαραίτητος. Αν δεν τον γράψεις θα γράψει έναν εννοούμενο ο μεταγλωττιστής· τα αντικείμενά σου θα ξεκινούν με μη καθορισμένη τιμή που, πιθανότατα, δεν συμμορφώνεται με την αναλλοίωτη. Συνήθως, μπορεί να δηλωθεί μαζί με τον δημιουργό με αρχικές τιμές («2 σε 1»).

Δημιουργός Αντιγραφής (ΔΑ, copy constructor): Καλείται α) όταν κατά τη δήλωση ενός αντικειμένου του δίνουμε ως αρχική τιμή την τιμή ενός άλλου της ίδιας κλάσης, β) για πέρασμα παραμέτρου τιμής σε συνάρτηση γ) επιστροφή τιμής από συνάρτηση με τύπο. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά αρκεί η αυτόματη αντιγραφή; Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Τελεστής Εκχώρησης (ΤΕ, "="): Εκχωρεί την τιμή μιας παράστασης σε μια μεταβλητή (αντικείμενο) του ίδιου τύπου. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά δεν είναι σίγουρο ότι κάνει σωστά τη δουλειά. Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Καταστροφέας (ΚΑ, destructor): Ο καταστροφέας είναι απαραίτητος για αντικείμενα που παίρνουν δυναμικά πόρους τη διάρκεια της ζωής τους, π.χ. μνήμη για έναν δυναμικό πίνακα. Δεν χρειάζεται να τον γράψεις όταν η αυτόματη καταστροφή του αντικειμένου κάνει σωστά τη δουλειά.

Συναρτήσεις *get* (GE) για όλα τα χαρακτηριστικά: Εξετάζουμε την ανάγκη για όλα τα χαρακτηριστικά αλλά δεν γράφουμε για όλα οπωσδήποτε! Π.χ. για την *BString* δεν υπάρχει λόγος να γράψουμε μια *getReserved()* που θα επιστρέφει την τιμή της *bsReserved*: αυτή η μεταβλητή είναι ένα «μυστικό» της υλοποίησης που κάνουμε. Να σημειώσουμε ότι, μερικές φορές, αυτές οι μέθοδοι δεν έχουν όνομα *getΚάτι*· παράδειγμα η *BString*::*length()* που είναι στην πραγματικότητα η *getLen* και η *BString*::*c_str()* που είναι η *getData*.

Συναρτήσεις *set* (SE) για όλα τα χαρακτηριστικά: Για όλα; Όχι! Για παράδειγμα, στη *BString* δεν είναι δυνατόν να επιτρέψουμε τη αυθαίρετη μεταβολή των *bsLen* και *bsReserved*. Οι συναρτήσεις *set* μεταβάλλουν τις τιμές μελών και οι νέες τιμές πρέπει να συμμορφώνονται με την αναλλοίωτη. Οι τιμές ορισμένων μελών μπορεί να αλλάζουν από άλλες μεθόδους σύμφωνα με τη λειτουργία του αντικειμένου.

Αν κάθε αντικείμενο έχει **πίνακα(-ες)** –ή συλλογές αντικειμένων άλλου είδους– τότε εξέτασε την ανάγκη για ύπαρξη μεθόδων για:

- **αναζήτηση**,
- **ανάκτηση**,
- **ενημέρωση** (εισαγωγή, διαγραφή, τροποποίηση)

ενός στοιχείου της (κάθε) συλλογής.

Στη *BString* μπορούμε να κάνουμε ανάκτηση και τροποποίηση με την επιφόρτωση του "`[]`". Το ότι αυτό φαίνεται πολύ βολικό και φυσικό για αυτήν την περίπτωση δεν σημαίνει ότι μπορεί να εφαρμοσθεί σε κάθε πίνακα.

Παρατηρήσεις:

1. Μερικοί προγραμματιστές θεωρούν ότι όταν έχουμε να γράψουμε συναρτήσεις *get* και *set* καλύτερα να μη γράψουμε κλάση αλλά δομή.[10] Ο δικός μας σχετικός κανόνας (§19.5) είναι λίγο διαφορετικός και βασίζεται στην αναλλοίωτη.[11] Τον υπενθυμίζουμε: **Αν μια κλάση**
   - **έχει αναλλοίωτη `true` (τα πάντα δεκτά) και**
   - **δεν έχει «μυστικά» της υλοποίησης, δηλαδή μέλη στα οποία δεν θέλουμε να δώσουμε άμεση πρόσβαση,**

   θα προτιμούμε να ομαδοποιούμε τα στοιχεία μας με μια δομή.
2. Πολλοί προγραμματιστές γράφουν καταστροφέα πάντοτε, ακόμη και αν δεν κάνει οτιδήποτε. Π.χ. θα έγραφαν: "`~Date() { }`".[12] Αυτό θα κάνουμε και εμείς από εδώ και πέρα.
3. Φαίνεται ότι ο *Δημιουργός Αντιγραφής*, ο *Τελεστής Εκχώρησης* και ο *Καταστροφέας* πάνε μαζί: ή έχεις να γράψεις και τα τρία (αν παίρνουμε πόρους του συστήματος) ή τίποτε (κανόνας των τριών). Έτσι, έχουν τα πράγματα συνήθως αλλά όχι πάντοτε. Θα δούμε και περιπτώσεις που χρειαζόμαστε μόνον κάποια από αυτά.
4. Φυσικά, η κλάση δεν περιορίζεται στα παραπάνω. Οι πιο ενδιαφέρουσες μέθοδοι θα προκύψουν από τις προδιαγραφές του προβλήματος ή τα σχέδια των εφαρμογών που θα χρησιμοποιήσουν την κλάση.

Στη συνέχεια θα εφαρμόσουμε τους παραπάνω κανόνες και θα ξαναγράψουμε μια κλάση από τα παλιά.

21.10.1 * Επιστροφή στις *string* και *BString*: Μέθοδος *reserve()*

Εδώ –και πριν δώσουμε το παράδειγμά μας– θα πρέπει να κάνουμε μια παρένθεση διότι όποιος «ερευνά τας γραφάς» γίνεται καχύποπτος: «*να μη γράψουμε getReserved() και setReserved(); Και γιατί η string έχει τις μεθόδους capacity() και reserve();*»

Η C++ δίνει –στον προγραμματιστή που χρησιμοποιεί τη *string*– τη δυνατοτητα να ελαχιστοποιήσει τις αλλαγές του δυναμικού πίνακα και –επομένως– τις αντιγραφές που

---

[10] Π.χ. στο (Lockheed Martin 2005) ο *AV Rule 123* λέει: "*The number of accessor and mutator functions* ***should*** *be minimized*", δηλαδή: *ο αριθμός των συναρτήσεων get και set πρέπει να ελαχιστοποιείται.*

[11] Να υπενθυμίσουμε ότι και στο (Lockheed Martin 2005) ο *AV Rule 66* λέει: "*A class* ***should*** *be used to model an entity that maintains an invariant.*"

[12] Συνηθέστατα: "`virtual ~Date() { }`"· θα το καταλάβεις αργότερα.

γίνονται σε κάθε αλλαγή. Πώς; Με το να καθορίσει το μεγεθος του πίνακα καλώντας τη μέθοδο *reserve()*. Δες ένα παράδειγμα:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1( "test text" );

   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;

   s1.reserve( 40 );
   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;
}
```

Αποτέλεσμα:

```
test text  9  9
test text  9  40
```

Η "`s1.capacity()`" μας επιστρέφει ως τιμή (`size_t`)[13] το μέγιστο μήκος που μπορεί να φτάσει το κείμενο που είναι αποθηκευμένο στην *s1* χωρίς να χρειαστεί να αλλάξει ο δυναμικός πίνακας. Με την "`s1.reserve(40)`" αλλάζουμε σε "`40`" (αυτό) το μέγιστο μήκος.

Η "`s1.reserve(` *n* `)`"

- Δεν θα αλλάξει την τιμή της *s1*.
- Αν *n* > *s1*`.`*capacity()* θα αλλάξει το μέγιστο μήκος σε *n* (ή μεγαλύτερο) εκτός αν δεν βρει αρκετή μνήμη ή *n* > *s1*`.`*max_size()*.
- Αν *n* < *s1*`.`*capacity()* δεν είναι απαραίτητο να αλλάξει κάτι.

Ας γράψουμε τις δύο αυτές μεθόδους για τη *BString*. Να παραδεχτούμε ότι η *capacity()* είναι η *getReserved()*:

```
   size_t capacity() const {  return bsReserved;  }
```

Η *reserve()* είναι πιο περίπλοκη:

```
void BString::reserve( int newRes )
{
   if ( newRes < 0 )
      throw BStringXptn( "reserve",
                         BStringXptn::domainError, newRes );
   if ( newRes > bsReserved ||
        (newRes < bsReserved/2 && newRes > bsLen) )
   {
      char* tmp;
      newRes = (newRes/bsIncr+1)*bsIncr;
      try {  tmp = new char[newRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "reserve", BStringXptn::allocFailed ); }
      bsReserved = newRes;
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
} // BString::reserve
```

Πρόσεξε ότι αλλάζουμε την τιμή του *bsReserved* αν:

- *newRes* > *bsReserved* ή
- *bsLen* < *newRes* < *bsReserved*/2

Ακόμη, για να διατηρήσουμε την αναλλοίωτή μας αλλάζουμε την τιμή της *newRes* σε "`(newRes/bsIncr+1)*bsIncr`".

---

[13] Ακριβέστερα: `string::size_type`.

Τέλος, πρόσεξε ότι αν η *reserve()* κληθεί με αρνητικό όρισμα ρίχνουμε εξαίρεση με κωδικό λάθους *BStringXptn*::*domainError* ενώ η *string*::*reserve()* ρίχνει (και) σε αυτήν την περίπτωση *bad_alloc*.[14]

Όπως βλέπεις, αυτήν τη μέθοδο δεν τη λές απλώς «*setReserved*»!

Αν προσαρμόσεις το παραπάνω προγραμματάκι στη *BString* θα πάρεις:

```
test text  9  16
test text  9  48
```

21.11 Λίστα με Απλή Σύνδεση

Στο Παράδ. 3 της §16.13 είδαμε μια *λίστα με απλή σύνδεση* (τύπος *SList*), με περιεχόμενο αντικείμενα τύπου *GrElmn*. Τώρα, θα ξαναλύσουμε το πρόβλημα γράφοντας την κλάση *SList* με εφαρμογή της συνταγής που είδαμε στην προηγούμενη παράγραφο.

Αναλλοίωτη (ΑΝ): Η λίστα είναι ένα σύνολο κόμβων που για τον καθένα –ας πούμε *aNode*– ισχύει το εξής:

είναι πρώτος κόμβος της λίστας (`*`*slHead* `==` *aNode*)[15]

ή (αποκλειστικώς, **xor**)

υπάρχει ένας και μόνον ένας άλλος κόμβος *bNode* που είναι προηγούμενος του *aNode*

(`*(`*bNode*`.`*lnNext*`) ==` *aNode*)[16]

Ακόμη:

υπάρχει ένας και μόνον ένας κόμβος που δεν έχει επόμενο και

τον δείχνει το *slTail* (*slTail*`->`*lnNext* `==` 0)

Αυτά ισχύουν για κάθε λίστα με απλή σύνδεση.

Για τη συγκεκριμένη λίστα:

- Το περιεχόμενο της είναι ένα *σύνολο* –ας το πούμε *content*– αντικειμένων τύπου *GrElmn*:
  - Σε κάθε κόμβο της λίστας, εκτός από τον `*`*slTail*, υπάρχει ένα και μόνον ένα στοιχείο του *content*. Αφού το *content* είναι σύνολο θα έχουμε:

    (*aNode* `!=` *bNode*) ⇒ (*aNode*`.`*lnData* `!=` *bNode*`.`*lnData*)
- Η σχέση *content* – κόμβων είναι ολική συνάρτηση (αλλά όχι "*επί*", αφού υπάρχει ο φρουρός στο `*`*slTail*).

Επομένως, η κλάση μας έχει μη τετριμμένη αναλλοίωτη.

Από τα παραπάνω γίνεται σαφές και κάτι άλλο: ο τύπος *ListNode* έχει θέση *μέσα* στην *SList*.

Έτσι, θα υλοποιήσουμε την κλάση μας ως:

```
class SList
{
public:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

[14] Δεν εξετάζουμε την περίπτωση: *n* > *max_size()*.

[15] Ή: *slHead* `==` `&`*aNode*.

[16] Ή: *bNode*`.`*lnNext* `==` `&`*aNode*.

Όπως καταλαβαίνεις, έξω από την κλάση ο τύπος του κόμβου θα πρέπει να γράφεται "`SList::ListNode`".

Ερήμην Δημιουργός (ΔΕ): Πρέπει να γράψουμε Ερήμην Δημιουργό για να βάλουμε τον φρουρό. Θα κρατήσουμε αυτόν που γράψαμε στην §16.13 με μια αλλαγή: Δεν θα ρίχνουμε εξαίρεση *ApplicXptn* αλλά *SListXptn*:

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

Δημιουργός Αντιγραφής (ΔΑ): Κάθε λίστα δεσμεύει πόρους –συγκεκριμένα: δυναμική μνήμη– του συστήματος. Επομένως χρειάζεται να γράψουμε εμείς τον (σωστό) δημιουργό αντιγραφής:

```
SList::SList( const SList& other )
{
   try
   {
      slHead = new ListNode;
      slTail = slHead;
      ListNode* p( other.slHead );
      while ( p != other.slTail )
      {
         slTail->lnData = p->lnData;
         slTail->lnNext = new ListNode;
         slTail = slTail->lnNext;
         p = p->lnNext;
      }
      slTail->lnNext = 0;
   }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
} // SList::SList
```

Και είναι σωστός αυτός ο δημιουργός αντιγραφής; Ας πούμε ότι έχουμε να αντιγράψουμε μια λίστα με 200 στοιχεία και παίρνουμε *bad_alloc* όταν προσπαθήσουμε να πάρουμε μνήμη για το 110το στοιχείο· με τον παραπάνω δημιουργό θα έχουμε διαρροή μνήμης που πήραμε για τα πρώτα 109 στοιχεία! Ας την ξαναγράψουμε πιο προσεκτικά:

```
SList::SList( const SList& other )
{
   try {  slHead = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
   slTail = slHead;
   ListNode* p( other.slHead );
   while ( p != other.slTail )
   {
      slTail->lnData = p->lnData;
      try {  slTail->lnNext = new ListNode;  }
      catch( bad_alloc& )
      {
         while ( slHead != slTail )
         {
            ListNode* p( slHead );
            slHead = slHead->lnNext;
            delete p;
         }
         delete slHead;
         throw SListXptn( "SList", SListXptn::allocFailed );
      }
      slTail = slTail->lnNext;
      p = p->lnNext;
```

```
   }
   slTail->lnNext = 0;
} // SList::SList
```

Όπως βλέπεις, στην "`catch`" το πρώτο που κάνουμε είναι να επιστρέψουμε τη μνήμη που πήραμε για τους κόμβους που ήδη αντιγράψαμε· μετά ρίχνουμε την εξαίρεση. Αυτός ο δημιουργός είναι σωστός και έχει *βασική εγγύηση ασφάλειας* ως προς τις εξαιρέσεις.

Καταστροφέας (ΚΑ): Αφού το κάθε αντικείμενο έχει δεσμεύσει δυναμική μνήμη πρέπει να γράψουμε καταστροφέα που θα την ανακυκλώνει:

```
SList::~SList()
{
   while ( slHead != slTail )
   {
      ListNode* p( slHead );
      slHead = slHead->lnNext;
      delete p;
   }
   delete slHead;
   slTail = slHead = 0;
} // SList::~SList
```

Εδώ βλέπεις έναν καταστροφέα που δεν έχει «μόνο μια γραμμή». Φυσικά, δεν είναι τίποτε άλλο από την *SList_deleteAll()*.

Τελεστής Εκχώρησης (ΤΕ): Αφού γράψαμε δημιουργό αντιγραφής και καταστροφέα θα πρέπει να γράψουμε και τελεστή εκχώρησης:

```
SList& SList::operator=( const SList& rhs )
{
   if ( &rhs != this )
   {
      SList tmp( rhs );
      this->swap( tmp );
   }
   return *this;
} // SList::operator=
```

όπου:

```
void SList::swap( SList& other )
{
   std::swap( slHead, other.slHead );
   std::swap( slTail, other.slTail );
} // SList::swap
```

Συναρτήσεις *get* (GE): «Καλώς εχόντων των πραγμάτων» δεν θα έπρεπε να δώσουμε συναρτήσεις "*get*". Πάντως, η *list* της STL δίνει παρόμοιες μεθόδους και αντί για *getHead()* και *getTail()* τις ονομάζει *begin()* και *end()* αντιστοίχως. Αυτό θα κάνουμε και εδώ. Τις ορίζουμε `inline`:

```
   ListNode* begin()  {  return slHead;  }
   ListNode* end()  {  return slTail;  }
```

Στο επόμενο κεφάλαιο θα δούμε και μια περίπτωση που μπορούμε να τις χρησιμοποιήσουμε αφού τις αλλάξουμε λίγο.

Συναρτήσεις *set* (SE): Φυσικά αποκλείεται να δώσουμε δικαίωμα για αλλαγή τιμών των δύο μελών αφού κάτι τέτοιο θα ήταν καταστροφικό για τη λίστα.

Διαχείριση Περιεχομένου: Θα γράψουμε μεθόδους διαχείρισης του περιεχομένου όπως περίπου δουλέψαμε και στη *Route*. Η διαχείριση του περιεχομένου θα γίνεται με βάση τον ατομικό αριθμό που τον επιλέξαμε ως κλειδί για τον *GrElmn*. Θα ξεκινήσουμε με την αντίστοιχη της *findNdx()*. Η συνάρτηση που θα γράψουμε θα επιστρέφει βέλος και όχι δείκτη αφού τώρα δεν έχουμε πίνακα:

```
SList::ListNode* SList::findPtr( int aAn ) const
{
   ListNode* fv;
   if ( aAn <= 0 )
```

```
      fv = slTail;
   else
   {
      slTail->lnData = GrElmn( aAn );
      fv = slHead;
      while ( (fv->lnData).getANumber() != aAn )  fv = fv->lnNext;
   }
   return fv;
} // SList::findPtr
```

Πρόσεξε τα εξής:

1. Η μέθοδος ψάχνει να βρει κόμβο που να έχει στο *lnData* τα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*. Αν βρει τέτοιο κόμβο επιστρέφει βέλος προς αυτόν· αλλιώς επιστρέφει βέλος προς τον φρουρό (== *slTail*).
2. Αν πάρει μη θετικό ακέραιο ως όρισμα δεν ρίχνειε εξαίρεση αλλά αναφέρει «δεν το βρήκα.»
3. Αυτή η μέθοδος είναι η αρχή της εγκατάλειψης της λειτουργίας της στοίβας που είχαμε στην αρχική λίστα. Σε μια «γνήσια» στοίβα δεν υπάρχει διαδικασία αναζήτησης· το μόνο που βλέπουμε είναι η κορυφή.

Αναζήτηση: Η *findPtr()*, όπως και η *findNdx()*, θα είναι **private**. Η αναζήτηση από ένα πρόγραμμα-πελάτη θα γίνεται με τη μέθοδο:

```
bool SList::find1Elmn( int aAn ) const
{
   return ( findPtr(aAn) != slTail );
} // SList::find1Elmn
```

που επιστρέφει "`true`" αν βρει στη λίστα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*.

Ανάκτηση: Και η ανάκτηση των δεδομένων για ένα στοιχείο γίνεται με βάση τον ατομικό αριθμό:

```
const GrElmn& SList::get1Elmn( int aAn ) const
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode == slTail )
      throw SListXptn( "get1Elmn", SListXptn::notFound, aAn );
   return ptrToNode->lnData;
} // SList::get1Elmn
```

Όπως κάναμε στην *get1RouteStop()* έτσι και εδώ, αν δεν βρούμε το στοιχείο που ζητείται, ρίχνουμε εξαίρεση. Αυτό σημαίνει ότι θα πρέπει να έχουμε εξασφαλίσει την ύπαρξη των δεδομένων πριν δώσουμε το μήνυμα *get1RouteStop()*.

Εισαγωγή: Η *push_front()* δεν μας εξασφαλίζει τη μοναδικότητα των αντικειμένων που περιέχει η λίστα. Γράφουμε λοιπόν την:

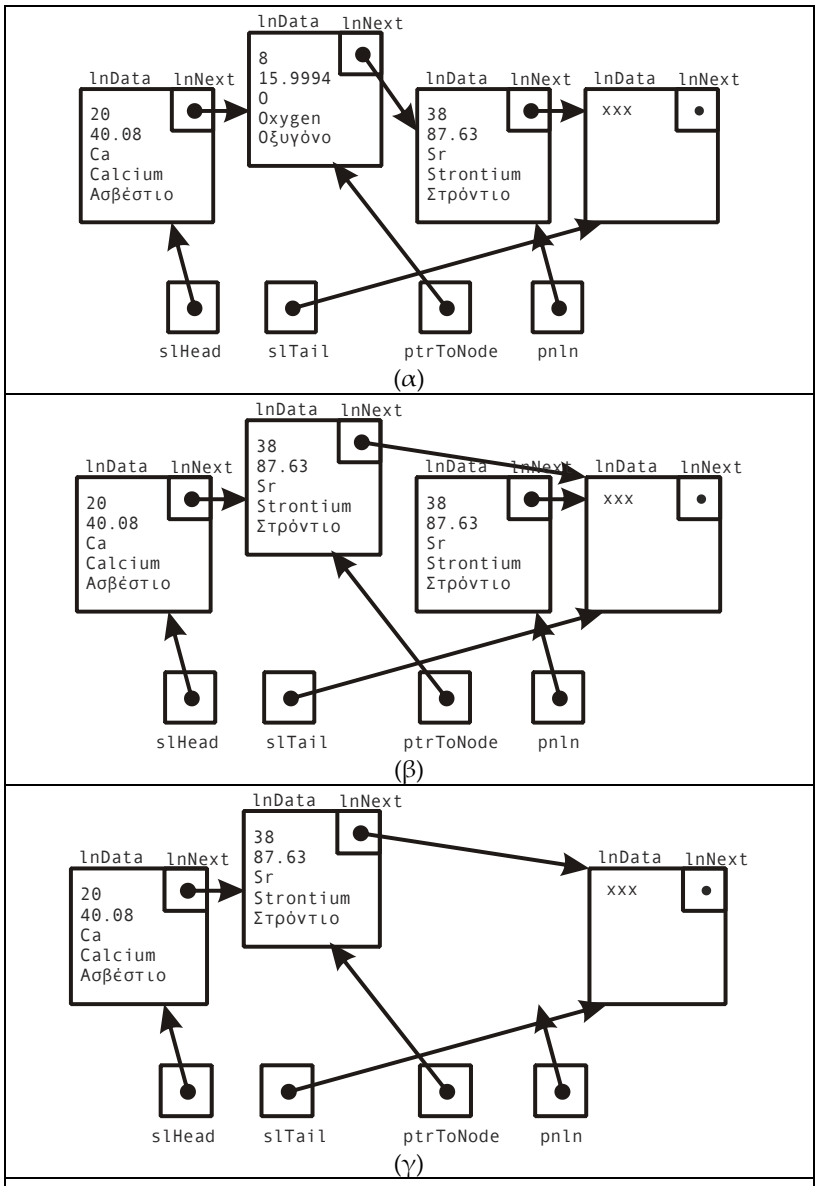
```
void SList::insert1Elmn( const GrElmn& aItem )
{
   ListNode* ptrToNode( findPtr(aItem.getANumber()) );
   if ( ptrToNode == slTail ) // δεν υπάρχει
   {
      push_front( aItem );
   }
} // SList::insert1Elmn
```

Αυτή καλεί την *push_front()* αφού έχει εξασφαλίσει ότι τα δεδομένα που θέλουμε να εισαγάγουμε δεν υπάρχουν ήδη στη λίστα. Για να αποτρέψουμε κακή χρήση της *push_front()* που θα παραβίαζε την αναλλοίωτη θα πρέπει να την «κρύψουμε» σε περιοχή **private**.

Όπως στην *insert1RouteStop()*, αν βρούμε το προς εισαγωγή στοιχείο στη λίστα δεν κάνουμε οτιδήποτε άλλο.

Διαγραφή: Εδώ έχουμε μεγάλη απόκλιση από τη λογική στοίβας: στη στοίβα η μόνη διαγραφή που μπορεί να γίνει είναι αυτή της κορυφής (*pop_front*). Εδώ, αν μας ζητηθεί να διαγράψουμε τα δεδομένα του στοιχείου με ατομικό αριθμό *aAn*, το αναζητούμε:

```
   ListNode* ptrToNode( findPtr(aAn) );
```



Σχ. 21-1 Τα τρία βήματα διαγραφής του κόμβου που δείχνει το βέλος *ptrToNode*. Στο (α) φαίνεται η κατάσταση μετά την εκτέλεση της εντολής "`ListNode* pnln(ptrToNode->lnNext)`". Στο (β) έχουμε την κατάσταση μετά την "`*ptrToNode = *pnln`". Στο (γ) φαίνεται η τελική κατάσταση μετά την "`delete pnln`".

και αν το βρούμε (*ptrToNode* `!=` *slTail*) πρέπει να διαγράψουμε τον κόμβο που δείχνει το *ptrToNode*.

Υπάρχει όμως ένα πρόβλημα: θα πρέπει να αλλάξουμε την τιμή του *lnNext* του *προηγούμενου* κόμβου που δεν ξέρουμε ποιος είναι! Θα κάνουμε κάτι άλλο:

```
ListNode* pnln( ptrToNode->lnNext );
*ptrToNode = *pnln;
delete pnln;
```

Δηλαδή: *αντιγράφουμε τον επόμενο κόμβο* (υπάρχει πάντοτε, να είναι καλά ο φρουρός) *στον στοχευόμενο και διαγράφουμε τον επόμενο*. Δες το Σχ. 21-1 όπου υποτίθεται ότι θέλουμε (α) να διαγράψουμε το "Οξυγόνο". Αντιγράφουμε (β) τα δεδομένα για το "Στρόντιο" σβήνοντας αυτά του Οξυγόνου. Αντιγράφουμε και το βέλος *lnNext* που τώρα πια δείχνει τον φρουρό. Τελικώς (γ) ανακυκλώνουμε τον κόμβο που ήταν το "Στρόντιο".

Αν ο κόμβος που ανακυκλώνουμε (*pnln) είναι ο φρουρός θα πρέπει να αλλάξουμε την τιμή του *slTail*. Στην περίπτωση αυτή δεν υπάρχει λόγος να αντιγράψουμε ολόκληρον τον κόμβο· αρκεί ο μηδενισμός του *lnNext*:

```
void SList::erase1Elmn( int aAn )
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode != slTail ) // υπάρχει
   {
      ListNode* pnln( ptrToNode->lnNext );
      if ( pnln != slTail )
         *ptrToNode = *pnln;
      else
      {
         ptrToNode->lnNext = 0;
         slTail = ptrToNode;
      }
      delete pnln;
   }
} // SList::erase1Elmn
```

Τροποποίηση: Ο πιο απλός τρόπος να αποφύγουμε παραβίαση της αναλλοίωτης από τροποποίηση δεδομένων ενός στοιχείου είναι αυτός που χρησιμοποιήσαμε και στη *RouteStop*:

```
      GrElmn tmp( lst.get1Elmn(n) );
      Τροποποίησε το tmp
      lst.erase1Elmn( n );
      lst.insert1Elmn( tmp );
```

21.11.1 Άλλες Μέθοδοι;

Αυτά που είδαμε μέχρι εδώ ήταν αυτά που βγαίνουν από τη συνταγή μας. Υπάρχουν άλλες ανάγκες; Ναι, η φύλαξη των δεδομένων που έχουμε αποθηκευμένα στη λίστα. Στο αρχικό πρόγραμμα, για τη φύλαξη, καλούσαμε τη συνάρτηση *saveUpdateList()*. Μπορούμε να την τροποποιήσουμε και να τη χρησιμοποιήσουμε και τώρα. Η τροποποίηση έχει να κάνει με το ότι έχουμε «κρύψει» τα δυο μέλη της κλάσης:

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

Ο τύπος της *p* (μεταβλητή-βέλος) με την οποία διασχίζουμε τη λίστα είναι "`const SList::ListNode*`". Αντί για *lst.slHead*, *lst.slTail* έχουμε `lst.begin()`, `lst.end()` αντιστοίχως.

«Καλά», θα πεις, «εδώ προσπαθούμε να κρύψουμε τις λειτουργίες της λίστας και τώρα θα πρέπει να τη διασχίσουμε σε εξωτερική συνάρτηση;! Δεν γίνεται να κάνουμε τη φύλαξη με μια μέθοδο;» Σωστή παρατήρηση! Αλλά σκέψου το εξής: Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. Εδώ, για παράδειγμα, τα αντικείμενα που αποτελούν το περιεχόμενο της λίστας φυλάγονται σε αρχείο τυχαίας πρόσβασης και έχουν ένα μέλος που καθορίζει τη θέση τους μέσα στο αρχείο. Καταλαβαίνεις λοιπόν ότι η διάσχιση της λίστας δεν μπορεί να «κρυφτεί» σε κάποια μέθοδο. Στο επόμενο κεφάλαιο θα δούμε κάποια εργαλεία για να κάνουμε τη διάσχιση χωρίς να βλέπουμε τη «λεπτομέρεια» "`p = p->lnNext`".

Παρ' όλα αυτά, περισσότερο για λόγους επίδειξης, ας κάνουμε μια προσπάθεια να γράψουμε μια μέθοδο που να κάνει αυτή τη δουλειά:

```
void SList::save( ostream& bout,
```

```
                void (*wrProc)(const GrElmn&, ostream&) ) const
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      (*wrProc)( p->lnData, bout );
} // save
```

Εδώ έχουμε μια μέθοδο που διασχίζει τη λίστα και για κάθε κόμβο καλεί μια συνάρτηση που περνούμε ως παράμετρο. Η κλήση της γίνεται ως εξής:

```
      lst.save( bInOut, writeRandom );
```

Είναι καλύτερη αυτή η λύση; Όπως νομίζεις... Πλεονεκτεί στο ότι έκρυψε τη διάσχιση της λίστας, μειονεκτεί στο ότι περνούμε μια εξωτερική συνάρτηση ως παράμετρο σε μια μέθοδο της κλάσης.

Στη συνέχεια θα προτιμήσουμε τη *saveUpdateList()*.

21.11.2 Το Πρόγραμμα

Το πρόγραμμα τώρα θα είναι ως εξής:

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

#include "GrElmn.cpp"
#include "SList.cpp"

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

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

void saveUpdateList( fstream& bInOut, const SList& lst );
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

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

   try
   {
      SList   lst;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            GrElmn tmp;
            if ( lst.find1Elmn(aa) )
            {
               tmp = lst.get1Elmn( aa );
```

```
               lst.erase1Elmn( aa );
            }
            else
               readRandom( tmp, bInOut, aa );
            editGrNameMM( tmp );
            lst.insert1Elmn( tmp );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
   }
   catch( GrElmnXptn& x )
   // . . .
   catch( SListXptn& x )
   // . . .
   catch( ApplicXptn& x )
   // . . .
} // main
```

Πρόσεξε ότι με την εγκατάλειψη της *elmntInList* το κομμάτι

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

αντικαταστάθηκε από το

```
            GrElmn tmp;
            if ( lst.find1Elmn(aa) )
            {
               tmp = lst.get1Elmn( aa );
               lst.erase1Elmn( aa );
            }
            else
               readRandom( tmp, bInOut, aa );
            editGrNameMM( tmp );
            lst.insert1Elmn( tmp );
```

21.12 * Βέλος προς Μέθοδο

Επιστρέφουμε στη μέθοδο *SList::save* με την εξής ερώτηση: Η *GrElmn* έχει τρεις μεθόδους – τις *save(), display()* και *writeToTable()*– με τις οποίες φυλάγουμε την τιμή του αντικειμένου σε κάποιο αρχείο. Είναι δυνατόν, με χρήση της *SList::save()*, να φυλάγουμε το περιεχόμενο της λίστας με οποιονδήποτε από τους τρεις τρόπους; Ναι, αλλά με την προϋπόθεση ότι θα γράψουμε την κατάλληλη εξωτερική συνάρτηση-κέλυφος.

Ας πάρουμε τη *GrElmn::display()*. Αυτή έχει μια παράμετρο τύπου *ostream&*. Όταν καλούμε την *SList::save* πρέπει να βάλουμε ως δεύτερο όρισμα μια συνάρτηση **void** με δύο παραμέτρους: η πρώτη είναι τύπου "`const GrElmn&`" και η δεύτερη τύπου *ostream&*. Η *GrElmn::display()* δεν ταιριάζει! Γράφουμε λοιπόν μια εξωτερική συνάρτηση:

```
void exDisplay( const GrElmn& aElmn, ostream& outF )
{
   aElmn.display( outF );
} // exDisplay
```

Αυτή κάνει τη δουλειά που θέλουμε και ταιριάζει ως δεύτερο όρισμα όταν καλούμε την *SList::save()*.

Αυτή φυσικά δεν είναι μια καλή λύση: Για να έχουμε αυτό που θέλουμε θα πρέπει να γράψουμε τρεις εξωτερικές συναρτήσεις, «ενδιάμεσες» μεταξύ *SList* και *GrElmn*.

Αυτό που θέλουμε είναι να γράψουμε μια:

```
void SList::toFile( ostream& out, OutFunc outProc ) const
```

```
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      ((p->lnData).*outProc)( out );
} // toFile
```

που όταν καλείται το *outProc* θα είναι βέλος προς τη *save* ή τη *display* ή τη *writeToTable*. Θέλουμε δηλαδή να χρησιμοποιήσουμε της τεχνική *συναρτήσεων ανάκλησης* (callback, §14.4) για μεθόδους. Πώς θα ορίσουμε τον τύπο *OutFunc*; Οι δηλώσεις των τριών μεθόδων στην *GrElmn* είναι:

```
class GrElmn
{
// . . .
   void save( ostream& bout ) const;
   void display( ostream& tout ) const;
   void writeToTable( ostream& tout ) const;
// . . .
}; // GrElmn
```

Να πούμε:

```
typedef void (*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc* δείχνει μια συνάρτηση `void` και **const** με μια παράμετρο τύπου *ostream*&; Όχι! Ο ορισμός πρέπει να είναι λίγο διαφορετικός:

```
typedef void (GrElmn::*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc δείχνει μια μέθοδο της GrElmn* **void** και **const** με μια παράμετρο τύπου *ostream*&.

Βάζουμε αυτόν τον ορισμό μετά τον ορισμό της *GrElmn* αλλά πριν τον ορισμό της *SList* και οι εντολές:

```
      ofstream tout( "test.txt" );
      lst.toFile( tout, &GrElmn::display );
      tout.close();
```

θα γράψουν το περιεχόμενο της λίστας στο αρχείο test.txt.

21.13 Μετατροπές Τύπου

Στα παραδείγματά μας μέχρι τώρα είδαμε μερικές περιπτώσεις μετατροπών τύπου:

- Ο δημιουργός `BString(const char* rhs)` παίρνει ένα κείμενο σε μορφή πίνακα χαρακτήρων και μας δίνει έναν αντικείμενο κλάσης *BString* με τιμή το ίδιο κείμενο.
- Αντιστρόφως, η *c_str()* μας δίνει ένα βέλος (`const char*`) προς πίνακα χαρακτήρων (με `'\0'` στο τέλος.)
- Είδαμε τον δημιουργό της *Date* να παίρνει τον φυσικό 2008 και να τον μετατρέπει σε αντικείμενο (01.01.2008)

Επειδή τέτοιες μετατροπές είναι, γενικώς, χρήσιμες οι προγραμματιστές της C++ έχουν παγιώσει την εξής συνταγή:

- Οι μετατροπές τύπου προς μια κλάση γίνονται από δημιουργούς της κλάσης.
- Οι μετατροπές τύπου από μια κλάση προς άλλους τύπους γίνεται με ειδικές συναρτήσεις μέλη της κλάσης.

Φυσικά, κανείς δεν σε εμποδίζει να παραβείς αυτούς τους κανόνες αρκεί να το κάνεις σωστά.

21.13.1 Μετατροπή με Δημιουργό

Λέγαμε πιο πριν (§21.1.1): «*Υπάρχει περίπτωση να έχουμε δημιουργό που χρειάζεται οπωσδήποτε κάποια αρχική τιμή στη δήλωση; Δεν είναι τόσο συχνή περίπτωση αλλά υπάρχει*

πιθανότητα να χρειαστείς κάτι τέτοιο.» Ο **δημιουργός μετατροπής** (conversion constructor) είναι ακριβώς μια τέτοια περίπτωση: ένας δημιουργός με μία παράμετρο που μετατρέπει μια τιμή του τύπου της παραμέτρου σε αντικείμενο της κλάσης που ανήκει. Για την ακρίβεια: ένας δημιουργός μετατροπής *θα πρέπει να μπορεί να κληθεί με ένα όρισμα μόνον*. Αυτό σημαίνει ότι μπορεί να έχει και άλλες παραμέτρους που όμως έχουν ερήμην τιμές.

Παραδείγματα ✎

Μετά τους διαχωρισμούς που κάναμε στην §21.1.1 έχουμε τον

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

που καλείται με ένα και μόνον ένα όρισμα και μετατρέπει μια τιμή "`const char*`"[17] σε αντικείμενο *BString* και έτσι –αν η *a* είναι τύπου *BString*– μπορούμε να δώσουμε:

```
a = BString( "abc" );
```

Έχουμε ακόμη τον

```
Date( int yp, int mp = 1, int dp = 1 );
```

που μπορεί να κληθεί με ένα όρισμα και να μετατρέψει μια τιμή "`int`" σε αντικείμενο *Date* και έτσι –αν η *d* είναι τύπου *Date*– με την εντολή:

```
d = Date( 375 );
```

δίνουμε στην *d* τιμή "01.01.375".

Στην §15.14.1 είδαμε τη συνάρτηση

```
GrElmn GrElmn_copyFromElmn( const Elmn& a )
```

–που από μια τιμή *Elmn* μας δίνει μια τιμή *GrElmn*– και υποσχεθήκαμε ότι «*αργότερα θα μάθουμε πώς μπορούμε να δώσουμε καλύτερη λύση.*» Τη δώσαμε στην §19.6 με τον δημιουργό μετατροπής

```
GrElmn::GrElmn( const Elmn& rhs )
```

✎✎✎

Παρατήρηση: ▶

Πρόσεξε ότι και οι «2 σε 1» δημιουργοί των *Date* και *BString* είναι δημιουργοί μετατροπής αφού είναι δυνατόν να κληθούν με ένα όρισμα.◀

Τώρα πρόσεξε το εξής: παρ' όλο που δεν έχουμε ορίσει

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

η εκχώρηση

```
a = "abc";
```

εκτελείται χωρίς κανένα πρόβλημα. Γιατί; Διότι καλείται αυτομάτως ο δημιουργός μετατροπής και μετατρέπει το `"abc"` σε αντικείμενο κλάσης *BString*. Δηλαδή και αυτή η εντολή εκτελείται ως "`a = BString("abc")`". Αυτό είναι καλό και βολικό.

Για τη *Date* όμως τα πράγματα αλλάζουν. Αν γράψεις

```
if ( d < 2012 )
{ . . .
```

γίνεται αυτομάτως η σύγκριση "`d < Date(2012)`" που υπολογίζεται ως «η *d* είναι *πριν από* την 01.01.2012» και είναι ακριβώς αυτό που θέλουμε. Αν όμως γράψουμε "`d > 2012`" αυτό θα υπολογιστεί ως «η *d* είναι *μετά* την 01.01.2012» ενώ εμείς θέλουμε να πούμε «η *d* είναι *μετά* την 31.12.2012». Ένα τέτοιο λάθος στο πρόγραμμα είναι πολύ δύσκολο να εντοπιστεί.

---

[17] Μπορούμε να τον κάνουμε να μετατρέπει ένα τμήμα του ορίσματος. Με τον:

```
BString( const char* rhs, int n=0 );
```

δημιουργούμε αντικείμενο που έχει αρχική τιμή το κομμάτι του *rhs* από τη θέση *n* και μετά. Με την "`a = BString("abcde", 2)`" η *a* θα πάρει ως τιμή `"cde"`.

Η C++ μας δίνει τη δυνατότητα να αποφύγουμε τέτοια λάθη. Αλλάζουμε τη δήλωση του δημιουργού μετατροπής:

```
   explicit Date( int yp, int mp = 1, int dp = 1 );
```

Αυτό το "**explicit**" έχει το εξής νόημα: «*μην κάνεις αυτόματες μετατροπές. Θα κάνεις μετατροπή αν ζητείται ρητώς.*» Δηλαδή δεν είναι δεκτές οι "`d = 375`", "`d < 2012`", "`d > 2012`" και θα πρέπει να γραφούν ως "`d = Date( 375 )`", "`d < Date(2012)`", "`d > Date(2012)`". Έτσι, (υποτίθεται ότι) ο προγραμματιστής θα σκεφτεί καλύτερα αυτό που γράφει.

Παρομοίως, στη *GrElmn* καλό θα ήταν να γράψουμε:

```
   explicit GrElmn( int aan=0, float aaw=0,
                    string as="", string anm="", string agn="" );
```

ώστε να μη γίνεται ερήμην μας η μετατροπή του "`88`" σε αντικείμενο *GrElmn* που αναφέρεται στο ράδιο.

Και για τη *BString* θα μπορούσαμε να δηλώσουμε:

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

αλλά, όπως είπαμε, στην περίπτωση αυτή η αυτόματη μεταροπή μας βολεύει. Πάντως, γενικώς, καλό θα είναι να ακολουθείς τον *Κανόνα OBJ32* του (CERT 2009):[18]

- ***Εξασφάλισε ότι οι δημιουργοί που μπορεί να κληθούν με ένα όρισμα δηλώνονται*** `explicit`.

21.13.2 Συναρτήσεις Μετατροπής

Οι συναρτήσεις μετατροπής είναι συναρτήσεις-μέλη που μετατρέπουν την τιμή ενός αντικειμένου σε έναν άλλον τύπο. Όπως είπαμε και πιο πάνω, τέτοια δουλειά κάνει και η *c_str()*· η C++ όμως μας δίνει μια πάγια μορφή για τέτοιες συναρτήσεις.

Μια συνάρτηση που κάνει την ίδια δουλειά με τη *c_str()* ορίζεται ως εξής:

```
public:
// . . .
   operator const char*() const
   {  bsData[bsLen] = '\0';  return bsData;  };
```

Ας πούμε ότι έχουμε δηλώσει:

```
      BString q( "abc" );
      char z[30];
```

Όπως ξέρουμε, μπορούμε να γράψουμε:

```
      strcpy( z, q.c_str() );
      cout << z << endl;
```

Τώρα, με τη νέα συνάρτηση, μπορούμε να γράψουμε και:

```
      strcpy( z, q );
      cout << z << endl;
```

Πρόσεξε ότι στη *strcpy* χρησιμοποιούμε το όνομα του αντικειμένου μόνο.

Πάντως, κανείς δεν μας εμποδίζει να γράψουμε:

```
public:
// . . .
   operator const char*()
   {  bsData[bsLen] = '\0';  return bsData;  };
   operator size_t() {  return bsReserved;  };
```

και να το χρησιμοποιήσουμε:

```
      size_t n( q );
```

Βέβαια, δημοκρατία έχουμε και κάνεις ό,τι θέλεις, αλλά μια τέτοια χρήση της δυνατοτητας που μας δίνεται είναι ανόητη.

---

[18] OBJ32: *Ensure that single-argument constructors are marked* "`explicit`".

Τελικώς όμως: Πόσο καλό είναι το ότι μπορούμε και γράφουμε `strcpy(z, q)` αντί για `strcpy(z, q.c_str());` Ε, δεν είναι και καμιά μεγάλη κατάκτηση. Μπορούμε να ζήσουμε και χωρίς αυτήν τη συνάρτηση· ή *c_str()* είναι σαφώς προτιμότερη.[19]

21.14 Στατικά Μέλη Κλάσης

Ας πούμε τώρα ότι έχουμε το εξής πρόβλημα: θέλουμε να έχουμε τη δυνατότητα να ξέρουμε πόσα αντικείμενα κλάσης *Date* υπάρχουν στο πρόγραμμά μας. Τι κάνουμε; Μια ιδέα είναι να δηλώσουμε μια καθολική μεταβλητή που οι δημιουργοί θα αυξάνουν κατά 1 ενώ ο καταστροφέας θα την μειώνει κατά 1 (θα πρέπει να γράψουμε έναν καταστροφέα που θα κάνει μόνον αυτήν τη δουλειά). Αυτό όμως είναι πολύ άκομψο και καθόλου ασφαλές: μια καθολική μεταβλητή που έχει σχέση αποκλειστικά με την κλάση! Η C++ μας δίνει τη δυνατότητα να κάνουμε κάτι καλύτερο.

Μπορούμε να δηλώσουμε μια *στατική* μεταβλητή-μέλος με όνομα *objCnt* και της δίνουμε αρχική τιμή έξω από τη δήλωση της κλάσης:

```
class Date
{
public:
// ...
private:
   static size_t objCnt;
   unsigned int  dYear;
// ...
}; // Date

   size_t Date::objCnt = 0;
```

Δηλώνουμε έναν καταστροφέα:

```
   ~Date()  {  --objCnt;  };
```

και αλλάζουμε τους δημιουργούς:

```
Date::Date()
{
   dYear = 1;  dMonth = 1;  dDay = 1;
                                  // 1η Ιανουαρίου του έτους 1 μ.Χ.
   ++objCnt;
} // Date::Date

Date:: Date( int yp, int mp, int dp )
{
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   dYear = yp;
   if ( mp <= 0 || 12 < mp )
      throw DateXptn( "Date", DateXptn::monthErr, mp );
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
   ++objCnt;
} // Date::Date

Date::Date( const Date& d )
{
   dYear = d.dYear;  dMonth = d.dMonth;  dDay = d.dDay;
   ++objCnt;
```

[19] Άλλοι είναι πιο «κάθετοι». Στο (Lockheed-Martin 2005) ο *AV Rule* 177 λέει: *"User-defined conversion functions should be avoided"* δηλαδή: συναρτήσεις μετατροπής (τύπου) που ορίζονται από τον χρήστη πρέπει να αποφεύγονται!

```
} // Date::Date
```

Ακόμη, δηλώνουμε μια ανοικτή μέθοδο:

```
   static int getObjCnt()  {  return objCnt;  };
```

Αν τώρα δώσουμε:

```
   Date k1, k2, k3[5];
   cout << Date::getObjCnt() << endl;
   {
      Date s1( 1950, 2, 1 ), s2( 2000, 2, 3 ),
           s3( 1960, 3, 1 ), *c;
      cout << Date::getObjCnt() << endl;
      c = new Date [3];
      cout << Date::getObjCnt() << endl;
      delete [] c;
      cout << Date::getObjCnt() << endl;
   }
   cout << Date::getObjCnt() << endl;
```

θα πάρουμε:

```
7
10
13
10
7
```

Στην αρχή έχουμε 7 αντικείμενα κλάσης *Date*: πέντε του *k*3 και άλλα 2 από τις *k*1, *k*2. Στη συνέχεια, στη σύνθετη εντολή δηλώνουμε 3 αντικείμενα κλάσης *Date*: *s*1, *s*2, *s*3 και τα αντικείμενα γίνονται 10. Όταν πάρουμε με τον `new` άλλα 3 αντικείμενα έχουμε συνολικά 13. Στη συνέχεια ανακυκλώνουμε τη μνήμη (`delete`) που πήραμε για τον *c* και τα αντικείμενα ξαναμένουν 10. Στο τέλος της σύνθετης εντολής καλείται αυτομάτως ο κατάστροφέας και καταστρέφει τα *s*1, *s*2, *s*3· έτσι ξαναμένουμε με 7 αντικείμενα.

Ας κάνουμε τώρα μερικές παρατηρήσεις στο παράδειγμά μας:

1. Το (στατικό) μέλος *objCnt* δηλώνεται στην κλάση και παίρνει αρχική τιμή έξω από αυτήν.
2. Όπως καταλαβαίνεις από τον χειρισμό του, το *objCnt* είναι μέλος της κλάσης και όχι του κάθε αντικειμένου της.
3. Έτσι όταν αναφερόμαστε στο *objCnt* χρησιμοποιούμε επίλυση εμβέλειας ("`Date::objCnt`") και όχι επιλογή μέλους ("`.objCnt`").
4. Η μέθοδος *getObjCnt*, που χειρίζεται μόνον το μέλος *objCnt*, είναι, και αυτή, *στατική*. Και αυτή έχει σχέση με ολόκληρη την κλάση και, όπως βλέπεις αναφερόμαστε και σε αυτήν με το πρόθεμα "`Date::`".

♦ ***Γενικώς με το*** `static` ***δηλώνουμε μεταβλητές, μεθόδους ή και σταθερές που ανήκουν σε ολόκληρη την κλάση και όχι στο κάθε αντικείμενο ξεχωριστά.***

Και σταθερές; Ναι! Όπως ακριβώς δηλώσαμε το στατικό μέλος *objCnt* μπορούμε να δηλώσουμε και να δώσουμε τιμή σε σταθερά (`const`) μέλη οποιουδήποτε τύπου. Θα το δούμε στην επόμενη παράγραφο.

Στην §19.1.2 λέγαμε για τις βοηθητικές συναρτήσεις «*αργότερα ... θα τις διακοσμήσουμε και με ένα "*`static`*"*». Πράγματι, οι *lastDay()* και *isLeapYear()* επεξεργάζονται τα δεδομένα που τους περνούμε μέσω των παραμέτρων τους χωρίς να έχουν οποιαδήποτε κατ' ευθείαν σχέση με τα (μη στατικά) μέλη *dYear*, *dMonth*, *dDay*. Το ίδιο ισχύει και για τη *cStrLen* της *BString*. Από εδώ και πέρα, όλες αυτές θα τις δηλώνουμε "**static**"· το ίδιο θα κάνουμε και για όλες τις βοηθητικές συναρτήσεις που θα γράφουμε.

Αν ξαναγυρίσουμε στην Άσκ. 19-3 τώρα μπορούμε να δώσουμε την πιο καλή λύση για την *isValidDate()*· μια στατική συνάρτηση:

```
   static bool isValidDate( int ay, int am, int ad )
   {
      bool fv( true );
      try{  Date( ay, am, ad );  }
```

```
      catch( DateXptn& ) {  fv = false;  }
      return fv;
   } // isValidDate
```

που τη χρησιμοποιούμε κάπως έτσι:

```
   if ( Date::isValidDate(iy, im, id) ) . . .
```

21.15 «Σταθερά» Μέλη Κλάσης

Όπως είπαμε στην §19.1.1, η C++ επιτρέπει να βάλουμε το χαρακτηριστικό "`const`" σε ορισμένες μεθόδους. Τι σημαίνει; *Η μέθοδος που έχει το χαρακτηριστικό* `const` *δεν αλλάζει τις τιμές των μελών*. Για παράδειγμα, στη *BString* είχαμε δηλώσει:

```
// . . .
   BString& operator=( const BString& rhs );
   inline const char* c_str() const;
   inline size_t length() const;
   inline bool empty() const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   int compare( const BString& rhs ) const;
// . . .
```

αφού οι *c_str()*, *length()*, *empty()* και *compare()* δεν μεταβάλλουν τις τιμές μελών.

Γιατί να δηλώσουμε `const` κάποια μέθοδο;

- Για λόγους τεκμηρίωσης. Χωρίς να εξετάσουμε το τι κάνει η μέθοδος καταλαβαίνουμε ότι δεν μεταβάλλει τις τιμές των μελών.
- Για λόγους ασφάλειας. Αν προσπαθήσουμε να μεταβάλλουμε την τιμή κάποιου μέλους, ο μεταγλωττιστής θα μας επαναφέρει στην τάξη.

Μπορείς να δηλώσεις και σταθερά αντικείμενα, π.χ.:

```
   const BString aSt( "qwdfvbnh" );
```

Αν προσπαθήσεις να χειριστείς ένα τέτοιο αντικείμενο με μέθοδο που δεν είναι `const`, π.χ.:

```
   aSt,swap( bSt );
```

θα πάρεις μήνυμα λάθος από τον μεταγλωττιστή.

Υπάρχουν περιπτώσεις που θέλουμε να αλλάζουν οι τιμές ορισμένων μελών ακόμη και σε σταθερά αντικείμενα. Αν δηλώσεις τα μέλη αυτά ως "`mutable`" (μεταλλάξιμα) έχεις αυτήν τη δυνατότητα.

Μπορούμε να έχουμε μέλη-σταθερές; Ο πιο απλός τρόπος είναι αυτός που χρησιμοποιούμε στις κλάσεις εξαιρέσεων: δηλώνουμε μέσα στην κλάση έναν απαριθμητό τύπο με όσες σταθερές θέλουμε, π.χ.:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr, outOfLimits };
// . . .
}; // DateXptn
```

και χρησιμοποιούμε στο πρόγραμμα τις σταθερές *yearErr*, *monthErr*, *dayErr*, *outOfLimits* ως: `DateXptn::yearErr`, `DateXptn::monthErr`, `DateXptn::dayErr`, `DateXptn::outOfLimits`.

Πρόσεξε τώρα το παρακάτω παράδειγμα για να δεις τι άλλες δηλώσεις σταθερών μπορείς να κάνεις:

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

using namespace std;

class A
{
private:
```

```
// . . .
   static const int size = 10;
public:
// . . .
   static const double cx;
   static const string car;
   static const char   carc[size];
}; // A
   const double A::cx = 1.234;
   const string A::car = "123456789";
   const char  A::carc[A::size] = "123456789";

int main()
{
   cout << A::cx << "  " << A::car << "  " << A::carc << endl;
}
```

Μέσα στην κλάση μπορείς να ορίσεις *ακέραιες* σταθερές, όπως η *size*.

Σταθερές *άλλων τύπων*, όπως οι *cx*, *car*, *carc*, δηλώνονται μέσα στην κλάση αλλά ορίζονται έξω από αυτήν.

Όπως καταλαβαίνεις, με βάση τα παραπάνω, μπορούμε να γράψουμε:

```
class BString
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
public:
// ...
private:
   static const size_t bsIncr = 16;
   char*  bsData;
   size_t bsLen;
   size_t bsReserved;
}; // BString
```

21.16 «Σταθερά» Μέλη Αντικειμένου

Καλά τα σταθερά μέλη κλάσης έχουν νόημα· τα *σταθερά μέλη ενός μη-σταθερού αντικειμενου* τι νόημα έχουν; Ως απάντηση, θα ξαναθυμίσουμε αυτά που λέγαμε στο παράδειγμα της μπαταρίας (§19.7) «*δεν θα πρέπει να υπάρχουν μέθοδοι που να αλλάζουν τις τιμές των bVoltage και bMaxEnergy, αφού αυτά παριστάνουν σταθερά χαρακτηριστικά της μπαταρίας που της αποδίδονται με την κατασκευή* (*δημιουργία*) *της*.» Και ακόμη καλύτερο θα ήταν να δηλώναμε:

```
class Battery
{
public:
// . . .
private:
   const double bVoltage;   // volts
   const double bMaxEnergy; // joules
   double       bEnergy;    // joules
}; // Battery
```

Ωραία! Αυτό επιτρέπεται, αλλά *πώς θα δίνουμε τιμές στα bVoltage και bMaxEnergy* του *κάθε αντικειμένου που δηλώνουμε*; Από τη στιγμή που δημιουργούνται τα μέλη απαγορεύεται να αλλάξουμε την (όποια) τιμή τους Ο μόνος τρόπος είναι η *λίστα εκκίνησης* που ορίζει τις τιμές τους όταν δημιουργούνται. Και πάλι δηλώνουμε:

```
   Battery( double v = 12, double me = 5e6 );
```

αλλά τώρα ορίζουμε:

```
Battery::Battery( double v, double me )
   :  bVoltage( v ), bMaxEnergy( me )
{
   if ( v <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::voltageErr, v );
```

```
   if ( me <= 0 )
      throw BatteryXptn( "Battery", BatteryXptn::energyErr, me );
   bEnergy = bMaxEnergy;
} // Battery::Battery
```

Και στην κλάση για την πισίνα, οι διαστάσεις της πισίνας θα πρέπει να είναι σταθερές. Αφήνουμε ως άσκηση τις σχετικές αλλαγές.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

21-1 Όπως ξέρεις μπορούμε να δηλώσουμε:

```
   string os1( 7, '#' );
```

και η *os1* να πάρει ως τιμή το "**#######**". Εφοδίασε τη *BString* με το εργαλείο που χρειάζεται για να αποκτήσει αυτήν τη δυνατότητα.

Β Ομάδα

21-2 Γράψε μια μέθοδο *setToday()* για τη *Date* ώστε να δίνει στο αντικείμενο ως τιμή την τρέχουσα ημερομηνία από το ρολόι του υπολογιστή.

21-3 Θέλουμε ένα εργαλείο, ας το πούμε *today()*, της *Date* που θα μας δίνει την τρέχουσα ημερομηνία. Πώς θα μπορούσε να γίνει; Σκέψου το σε σχέση με το εργαλείο της προηγούμενης άσκησης αλλά και ανεξάρτητα από εκείνο.

κεφάλαιο

22

Επιφόρτωση Τελεστών

Ο στόχος μας σε αυτό το κεφάλαιο:

Να γνωρίσουμε μερικούς κανόνες για την επιφόρτωση τελεστών σε κλάσεις. Αυτοί είναι κάπως διαφορετικοί από αυτούς που ξέρουμε μέχρι τώρα· οι αλλαγές είναι απαραίτητες αφού στις κλάσεις έχουμε και κρυμμένα μέλη. Η δυνατότητα για επιφόρτωση με μεθόδους αξιοποιείται στον μέγιστο βαθμό.

Προσδοκώμενα αποτελέσματα:

Θα μπορείς να επιφορτώνεις οποιονδήποτε τελεστή σε οποιαδήποτε κλάση και να χρησιμοποιείς μερικές πάγιες τεχνικές επιφόρτωσης. Θα μπορείς να αποφασίσεις αν είναι σωστό να γίνει κάποια επιφόρτωση.

Έννοιες κλειδιά:

- *επιφόρτωση τελεστή*
- *συναρτήσεις **friend***
- *κλάσεις **friend***
- *προσεγγιστές*
- *τεχνική **pimpl***

Περιεχόμενα:

22.1 Επιφόρτωση Τελεστών: Τι Ξέρουμε Μέχρι Τώρα ... 740
22.2 Προβλήματα Συμβατότητας ... 741
22.3 Συναρτήσεις και Κλάσεις "`friend`" ... 743
22.4 Προειδοποιητική Δήλωση ... 744
22.5 Ενικοί Τελεστές ... 744
22.5.1 Προθεματικοί Ενικοί Τελεστές ... 745
22.5.2 Ενικοί Μεταθεματικοί Τελεστές ... 746
22.6 Μη-Αντιμεταθετικοί Δυαδικοί Τελεστές ... 746
22.6.1 Αντικείμενο Αριστερά ... 746
22.6.1.1 Ο Τελεστής "`[]`" για τη *BString* ... 747
22.6.1.2 Ο Τελεστής "+=" για τη *BString* ... 747
22.6.1.3 Ο Τελεστής "+=" για τη *Date* ... 749
22.6.1.4 * Ο Χρόνος στη C ... 749
22.6.1.5 * Υλοποίηση της *forward()* ... 750
22.6.1.6 * Ο Τελεστής "++" της *Date* (ξανά) ... 751
22.6.2 Ο Τελεστής "`()`" και η Χρήση του ... 751
22.6.2.1 * Μέλος - Περίγραμμα Συνάρτησης ... 754
22.6.3 Αντικείμενο Δεξιά ... 754
22.7 Αντιμεταθετικοί Τελεστές ... 755
22.7.1 Από τον "@=" στον "@" ... 756
22.7.2 Σύγκριση Ημερομηνιών ... 757
22.7.3 Οι Τελεστές Σύγκρισης της *BString* ... 757
22.7.3.1 * Η Σειρά Ταξινόμησης ... 760
22.8 Τελεστές για τη *Vector3* ... 760
22.9 Διασχίζοντας τη Λίστα με τον "++" - Προσεγγιστές ... 761

22.9.1 Επιφόρτωση του “->”....764
22.10 * Απόκρυψη Υλοποίησης – Τεχνική “pimpl”.... 764
Ερωτήσεις - Ασκήσεις 770
Α Ομάδα....770

Εισαγωγικές Παρατηρήσεις:
Είναι πολύ βολικό να επιφορτώνεις τελεστές, για μια κλάση που γράφεις, αντί να γράφεις συναρτήσεις. Ή, αν θέλεις, να επιφορτώνεις *και* τελεστές εκτός από τις συναρτήσεις. Ο λόγος;

- Όπως είδαμε ήδη με τις δομές, μπορείς να χρησιμοποιήσεις για τα αντικείμενα περιγράμματα συναρτήσεων (ή και κλάσεων, όπως θα δούμε στη συνέχεια). Π.χ. για να χρησιμοποιήσεις περίγραμμα συνάρτησης για αναζήτηση σε πίνακα με στοιχεία τύπου *T* συνήθως απαιτείται να έχουμε επιφορτώσει για τον *T* με τον τελεστή “`==`”, ενώ για να χρησιμοποιήσεις περίγραμμα συνάρτησης για ταξινόμηση απαιτείται τουλάχιστον ο “`<`”.
- Θυμάσαι τι κάνουν! Αντί να ψάχνεις μέσα στην κλάση να βρεις αν τη μέθοδο την έχεις πει *next* ή *nextDay* ή *epomenh* ή *epomeni* ή *epom* χρησιμοποιείς τον “`++`” και τελείωσες! Αυτό βέβαια με μια προϋπόθεση:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο, η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

Για πρώτη φορά μιλήσαμε για επιφόρτωση τελεστών στην §14.6 ενώ στα κεφάλαια 20 και 21 ασχοληθήκαμε διεξοδικώς με την επιφόρτωση του τελεστή εκχώρησης “`=`”.

Στο κεφάλαιο αυτό, που ασχολείται με την επιφόρτωση τελεστών για κλάσεις, θα συμπληρώσουμε (και θα τροποποιήσουμε) τις οδηγίες της §14.6.4.

22.1 Επιφόρτωση Τελεστών: Τι Ξέρουμε Μέχρι Τώρα

Όπως έχουμε μάθει, μπορούμε να επιφορτώσουμε οποιονδήποτε τελεστή εκτός από τους

“`::`”, “`.`”, “`.*`”, “`?:`”

Ο τρόπος που μάθαμε να κάνουμε την επιφόρτωση ήταν ένας: μια καθολική συνάρτηση με το όνομα “`operator@`” για τον τελεστή “`@`”. Αλλά για δες πώς επιφορτώσαμε τον “`==`” στην §15.5.1:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day == rhs.day );
}; // operator==( const Date
```

και πώς στην §19.1.4:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.getYear() == rhs.getYear() &&
            lhs.getMonth() == rhs.getMonth() &&
            lhs.getDay() == rhs.getDay() );
}; // operator==( const Date
```

Τι μεσολάβησε; Στο κεφάλαιο 19 «κρύψαμε» τα μέλη της *Date* σε περιοχή **private**! Είπαμε βέβαια ότι οι “*get*” δεν αυξάνουν τον υπολογιστικό χρόνο αφού αυτές οι μέθοδοι είναι **inline**. Τι γίνεται όμως στις περιπτώσεις που θα χρειαστούμε μέλη για τα οποία δεν γράφουμε τέτοιες μεθόδους;

Και κάτι ακόμη: Ας πούμε ότι, για να προχωρούμε στην επόμενη μέρα, θέλουμε να επιφορτώσουμε τον “`++`” για τη *Date*. Θα χρειαστούμε τη *lastDay()*· αλλά και αυτήν την «κρύψαμε» σε περιοχή **private**.

Υπάρχει λοιπόν πρόβλημα με τα «κρυμμένα» μέλη και οι λύσεις –εκτός από τις "*get*"– είναι δύο:

- Να επιφορτώσουμε τους τελεστές με μεθόδους.
- Να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου στην (καθολική) συνάρτηση που επιφορτώνει κάποιον τελεστή.

Θα δούμε και τις δύο περιπτώσεις. Αλλά, όπως θα δεις στην επόμενη παράγραφο, η «επιλεκτική αποκάλυψη μυστικών» είναι μια, κατ' αρχήν τουλάχιστον, επικίνδυνη τεχνική. Θα προσπαθούμε λοιπόν να κάνουμε τις επιφορτώσεις μας με μεθόδους ή με καθολικές συναρτήσεις που χρησιμοποιούν συναρτήσεις-μέλη "*get*" που είναι **inline**.

Είδαμε ήδη την επιφόρτωση τελεστή με μέθοδο: αυτήν του "=" για τη *BString*. Η επιφόρτωση έγινε με μια μέθοδο:

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

Όταν τη γράφαμε, παίρναμε υπόψη μας ότι στην εκχώρηση "`a = Π`" το *Π* γίνεται *rhs* ενώ `*this` είναι το *a*. Αν δηλαδή είχαμε υλοποίηση με καθολική συνάρτηση[1], σύμφωνα με τις αντιστοιχίες που είδαμε στην §19.1 και 19.1.1 θα είχαμε:

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

Ο δυαδικός τελεστής "=" έχει τα εξής χαρακτηριστικά:

- *Έχει αριστερά το αντικείμενο* που μας ενδιαφέρει (`*this`).
- *Δεν είναι αντιμεταθετικός*, δηλαδή δεν είναι δυνατόν να γράψουμε "`Π = a`" αντί για "`a = Π`".

Οποιονδήποτε τελεστή με αυτά τα χαρακτηριστικά (άλλα παραδείγματα: "`+=`", "`[]`" κλπ) τον επιφορτώνουμε με μέθοδο.

Με μέθοδο επιφορτώνουμε κάι κάθε ενικό τελεστή.

Αν ο τελεστής είναι *δυαδικός αντιμεταθετικός* η επιφόρτωση με μέθοδο δεν είναι καλή ιδέα. Ας πάρουμε τον "==" για τη *BString*: Αν τον επιφορτώσεις με μέθοδο:

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

η σύγκριση "`a == "qwerty"`" θα είναι εφικτή ενώ η "`"qwerty" == a`" είναι παράνομη αφού το `"qwerty"` δεν είναι τιμή *BString*. Εδώ η επιφόρτωση θα πρέπει να γίνει όπως ξέρουμε: με καθολική συνάρτηση.

Με καθολική συνάρτηση επιφορτώνονται και οι *δυαδικοί μη αντιμεταθετικοί* τελεστές με το αντικείμενο στο *δεξιό μέρος*.

22.2 Προβλήματα Συμβατότητας

Στην §14.6, μιλώντας για επιφόρτωση τελεστών για πρώτη φορά βάζαμε έναν βασικό κανόνα:

♦ ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***

πράγμα που λέμε με άλλα λόγια και στην εισαγωγή του παρόντος κεφαλαίου.

Η τήρηση αυτού του κανόνα είναι απαραίτητη για να διευκολυνόμαστε από τις επιφορτώσεις. Η παράβασή του αντί για διευκόλυνση προκαλεί σύγχυση και προβλήματα στον προγραμματιστή-χρήστη της κλάσης.

Το επόμενο πρόβλημα είναι αυτό που ήδη αντιμετωπίσαμε με τον τελεστή "=" για τη *BString* και τη μέθοδο *assign()*. Για την επιφόρτωση του "=" η γλώσσα επιβάλλει μέθοδο με επικεφαλίδα:

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

[1] *Ο μεταγλωττιστής δεν θα σου επιτρέψει* να επιφορτώσεις τον "=" με άλλον τρόπο

Η *assign()*, που θα πρέπει να έχει ακριβώς την ίδια λειτουργία θα πρέπει –κατά παράβαση άλλων κανόνων που έχουμε θέσει– να έχει ακριβώς την ίδια επικεφαλίδα:

```
BString& assign( const BString& rhs );
```

Και μετά; Αντιγράφουμε το σώμα της πρώτης συνάρτησης ως σώμα της δεύτερης; Όπως είπαμε, η αντιγραφή είναι προσωρινή μόνον εγγύηση ταυτόσημης λειτουργίας. Η ταυτόσημη λειτουργία μπορεί να χαθεί όταν, αργότερα, θα χρειαστεί να τροποποιήσεις τις μεθόδους.

Η μοναδική εγγύηση είναι το να γράψουμε τον κώδικα μια φορά μόνον –είτε στην `operator=()` είτε στην *assign()*– και η δεύτερη επιφόρτωση να καλεί την πρώτη. Στην §20.4.1 ορίσαμε την `operator=` και ορίσαμε –**inline**– την *assign()* ως:

```
BString& assign( const BString& rhs ) { return (*this = rhs); }
```

Έτσι, αν έχουμε δηλώσει:

```
BString a, b;
```

η εντολή:

```
b.assign( a );
```

θα μετατραπεί σε "`b = a`".

Αν προτιμούσαμε να ορίσουμε την *assign()*, θα έπρεπε να ορίσουμε –και πάλι **inline**–

```
BString& operator=( const BString& rhs )
{  return ( this->assign(rhs) );  }
```

Σε μια τέτοια περίπτωση, η εντολή:

```
b = a;
```

θα μετατραπεί σε "`b.assign( a )`".

Είναι φανερό ότι –και στις δύο περιπτώσεις– το πρόγραμμά μας δεν «πληρώνει» οποιοδήποτε κόστος κλήσης συνάρτησης.

Όπως θα δεις στη συνέχεια, με τον ίδιο τρόπο θα αντιμετωπίσουμε το ζεύγος "`+=`", *append()* –για τη *BString*– και το ζεύγος "`+=`", *forward()*, για τη *Date*.

Γενικώς λοιπόν:

- ***Αν θέλουμε δύο μέθοδοι, ας πούμε a και b, να εκτελούν την ίδια ακριβώς λειτουργία ορίζουμε τη μια από αυτές –έστω την a– και μετά ορίζουμε τη b με κλήση της a.***

Ένα άλλο πρόβλημα συμβατότητας είναι αυτό των τελεστών με σχετικό νόημα, π.χ.: "`+`" και "`+=`". Αυτοί οι δύο τελεστές επιφορτώνονται για τη *BString* ώστε να κάνουν –όπως και στη *std::string*– σύνδεση δύο ορμαθών. Η "`a + b`" μας δίνει έναν νέο ορμαθό που αποτελείται από το κείμενο του *a* και στη συνέχεια το κείμενο του *b* χωρίς να αλλάζουν οι τιμές των *a* και *b*. Η "`a += b`" επισυνάπτει στο κείμενο του *a* το κείμενο του *b* αλλάζοντας την τιμή του *a*. Όπως θα δεις, η επιφόρτωση του "`+`" γίνεται με κλήση του "`+=`"· δηλαδή ο ορισμός της πράξης γράφεται μια φορά μόνον, για τον "`+=`".

Με τους "`+`" και "`-`" το πρόβλημα μπορεί να είναι ευρύτερο. Για παράδειγμα, για τη *Date* επιφορτώνουμε τον "`+`"

`+`: *Date* × `int` ↛ *Date*

με το εξής νόημα: *η "`d1 = d + n`" δίνει στη d1 ως τιμή ημερομηνία μεταγενέστερη κατά n μέρες της τιμής της d.* Αν τώρα θέλουμε να επιφορτώσουμε τον "`+=`" θα πρέπει να το κάνουμε έτσι που η "`d += n`" να έχει το ίδιο νόημα με την "`d = d + n`"· οτιδήποτε άλλο θα ήταν παραπλανητικό για τον προγραμματιστή-χρήστη της κλάσης. Τα ίδια ισχύουν και για τον "`++`": το "`++d`" δεν μπορεί παρά να σημαίνει "`d = d + 1`" ή "`d += 1`". Όπως θα δεις στη συνέχεια, ορίζουμε αρχικώς τη *forward*, στη συνέχεια επιφορτώνουμε τον "`+=`" ως ταυτόσημο με αυτήν και από τον "`+=`" ορίζουμε τις επιφορτώσεις των "`+`" και "`++`". Ο μεταθεματικός "`++`" επιφορτώνεται με κλήση του προθεματικού. Η άσκ. 22-1 σου ζητάει να συνεχίσεις, με την ίδια λογική, με τους "`-`", "`--`" και "`-=`".

- ***Αν έχεις να επιφορτώσεις για μια κλάση τους "+", "+=", "++", "-", "-=", "—" όρισε έναν από αυτούς και όρισε με κλήσεις σε αυτόν τις επιφορτώσεις των άλλων.***

Παρόμοια ισχύουν και τους "`*`", "`*=`","`/`", "`/=`" κ.ο.κ.

Θα πει κάποιος: «Και τι θα γίνει αν εγώ επιφορτώνω τον κάθε τελεστή με όποιο νόημα μου έλθει;» Δεδομένου του δημοκρατικού πολιτεύματος στο οποίο ζούμε, είναι σίγουρο ότι δεν θα πάει στη φυλακή! Αυτό που δεν είναι σίγουρο είναι ότι θα βρίσκει πελάτες για να πουλάει το λογισμικό που γράφει...

22.3 Συναρτήσεις και Κλάσεις "`friend`"

Ας δούμε τώρα πώς μπορούμε να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου σε μια καθολική συνάρτηση.

Η C++ μας δίνει όμως την εξής δυνατότητα: να δηλώσουμε μέσα σε μια κλάση, ας πούμε τη *Date*, ως **φίλη** (friend) μια συνάρτηση (τελεστή), ας πούμε την `operator==()`:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs );
public:
   Date();
// . . .
```

Με ποιο αποτέλεσμα;

- ***Αν μια συνάρτηση δηλωθεί ως `friend` (φίλη) μέσα σε μια κλάση, έχει τη δυνατότητα να «βλέπει» τα εσωτερικά μέλη των αντικειμένων της κλάσης.***

Έτσι, μπορούμε να γράψουμε, όπως παλιά:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
```

Με αυτόν τον τρόπο γλυτώσαμε από κλήσεις συναρτήσεων. Για την ακρίβεια δεν τις βλέπουμε πια· διότι –όπως είπαμε– και οι κλήσεις των "*get*" εξαφανίζονταν από τον μεταγλωττιστή λόγω του "`inline`".

Για μικρές καθολικές συναρτήσεις μπορείς, αν θέλεις, να βάλεις ολόκληρο τον ορισμό μαζί με τη δήλωση:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
public:
   Date();
// . . .
```

Στην περίπτωση αυτήν η συνάρτηση (στο παράδειγμά μας: `operator==()`) ορίζεται "`inline`".

Εκτός από συναρτήσεις είναι δυνατόν να δηλώσουμε και *φίλες κλάσεις*. π.χ.:

```
class A
{
friend class B;
public:
// ...
```

Όλες οι μέθοδοι ενός αντικειμένου κλάσης *B* που δηλώνεται ως "`friend`" μέσα στην κλάση *A*, έχουν τη δυνατότητα να βλέπουν τα εσωτερικά μέλη των αντικειμένων της κλάσης *A*. Αλλά:

♦ ***Η φιλία δεν είναι μεταβατική.***

Δηλαδή: το ότι η κλάση *B* δηλώνεται ως "`friend`" της *A* και μια συνάρτηση ή κλάση δηλώνεται ως "`friend`" της *B* δεν σημαίνει ότι είναι και φίλη της *A*.

Όπως καταλαβαίνεις, αυτό το «άνοιγμα» των «μυστικών» ενός αντικειμένου σε συναρτήσεις και –πολύ περισσότερο– σε κλάσεις δεν είναι ακίνδυνο. Όπως σου λέει και η πείρα σου, μερικές φορές φτάνει ένα "`==`" να γίνει "`=`" για να γίνει το «κακό». Για τον λόγο αυτόν θα χρησιμοποιούμε τις φιλίες όσο γίνεται λιγότερο και με τη μεγαλύτερη δυνατή προσοχή.

Ειδικώς για τις φιλίες με κλάσεις, ο κίνδυνος μπορεί να μετριασθεί αν δηλώσεις "`friend`" όχι ολόκληρη την κλάση αλλά συγκεκριμένη (-ες) μέθοδο (-ους) της. Ας πούμε ότι έχεις:

```
class Date;

class K
{
public:
   K( int aa=5 )  {  a = aa;  };
   int f( const Date& d ) const;
   int g( const Date& d ) const;
private:
   int a;
}; // K
```

και δηλώσεις:

```
class Date
{
friend int K::f( const Date& d ) const;
//  . . .
```

μπορείς στον ορισμό της *K*`::`*f* να χρησιμοποιείς τα κρυμμένα μέλη της παραμέτρου *d* ενώ αυτό απαγορεύεται στον ορισμό της *K*`::`*g*.

Εκείνο το "`class Date;`" στην αρχή τί είναι; Διάβασε παρακάτω...

22.4 Προειδοποιητική Δήλωση

Για να μπορέσουμε να κάνουμε τη δήλωση

```
class K
{
// . . .
   int f( const Date& d ) const;
// . . .
```

στο προηγούμενο παράδειγμα θα πρέπει ο μεταγλωττιστής να ξέρει τη *Date*. Δεν χρειάζεται να την ξέρει πλήρως· αρκεί μια **προειδοποιητική δήλωση** (forward declaration)

```
   class Date;
```

που –μοιάζει με τις επικεφαλίδες συναρτήσεων που βάζουμε πριν από την κλήση τους και– λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

22.5 Ενικοί Τελεστές

Δύο από τις οδηγίες της §14.6.4 που θα αλλάξουμε είναι αυτές για την *επιφόρτωση ενικών τελεστών για κλάσεις*.

♦ ***Οι ενικοί τελεστές για τις κλάσεις θα επιφορτώνονται με μεθόδους.***

22.5.1 Προθεματικοί Ενικοί Τελεστές

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν προθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

  Τώρα συμπληρώνουμε, αν *C* είναι μια κλάση:

- Αν ο *T* είναι *C*& η επιφόρτωση μπορεί θα γίνεται με μέθοδο (χωρίς παραμέτρους):

  `Trv operator@( )`

  Αν ο *T* είναι *C* ή `const` *C*& η επιφόρτωση μπορεί θα γίνεται με μέθοδο:

  `Trv operator@( ) const`

Παράδειγμα ✎

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον *προθεματικό* "`++`" για την *Date*. Τι θα κάνει; Θα προχωράει την ημερομηνία κατά μια ημέρα. Μόνον αυτό; Για να είμαστε συνεπείς με τη φιλοσοφία της C++ (C) θα πρέπει να επιστρέφει τη νέα ημερομηνία. Σύμφωνα με τις οδηγίες της §14.6.4 θα έπρεπε η επιφόρτωση να είναι της μορφής:

```
Date& operator++( Date& a )
```

αλλά σύμφωνα με την παραπάνω συμπλήρωση θα πρέπει να γίνει με μέθοδο:

```
Date& Date::operator++()
{
   if ( dDay < lastDay(dYear, dMonth) )
      ++dDay;
   else // αλλαγή μήνα
   {
      dDay = 1;
      if ( dMonth < 12 )
         ++dMonth;
      else // αλλαγή έτους
      {
         dMonth = 1;
         ++dYear;
      } // if ( dMonth...
   } // if ( dDay...
   return *this;
} // Date::operator++
```

Πως τον χρησιμοποιούμε;

```
Date d1( 2011, 3, 31 );
cout << d1 << endl;
++d1;
cout << d1 << endl;
```

Αντί για "`++d1`" μπορείς να γράψεις:

```
d1.operator++();
```

αλλά γιατί να το κάνεις;

Αν θελήσεις να επιφορτώσεις τον "`++`" με καθολική συνάρτηση (με μια παράμετρο) θα πρέπει να τη δηλώσεις "`friend`" οπωσδήποτε για να μπορέσεις να χρησιμοποιήσεις τη *lastDay*, που είναι **private**.

Παρατήρηση: ▶

Μετά αυτά που είπαμε στην §22.2, καταλαβαίνεις ότι με τον "`++`" της *Date* δεν τελειώσαμε...◀

✎✎✎

Αργότερα θα επιφορτώσουμε και άλλους προθεματικούς ενικούς τελεστές.

22.5.2 Ενικοί Μεταθεματικοί Τελεστές

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν μεταθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a, int )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

και τώρα συμπληρώνουμε: αν *C* είναι μια κλάση

- Αν ο *T* είναι *C*& η επιφόρτωση μπορεί να γίνει με μέθοδο της *C*:

  `Trv operator@( int )`

  Αν ο *T* είναι *C* ή `const` *C*& η επιφόρτωση μπορεί να γίνει με μέθοδο:

  `Trv operator@( int ) const`

Παράδειγμα ✎

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" για τη *Date*. Ο *T* σίγουρα είναι ο *Date*&. Επομένως συζητούμε για την περίπτωση:

`Trv operator++( int )`

Ο *Trv* ποιος θα είναι; Ο *Date* ή ο *Date*&; Ο τελεστής αυτός αλλάζει την τιμή του αντικειμένου αλλά επιστρέφει την προηγούμενη τιμή του. Αυτή θα πρέπει να έχει κρατηθεί σε μια *τοπική μεταβλητή της συνάρτησης*. Άρα δεν είναι δυνατόν να επιστρέψει τιμή τύπου *Date*&. Επομένως θα έχουμε:

`Date operator++( int )`

Κατά τα άλλα, η υλοποίηση είναι πολύ απλή αν χρησιμοποιήσουμε τον προθεματικό "`++`":

```
Date Date::operator++( int )
{
   Date sv( *this );
   ++(*this);
   return sv;
} // Date::operator++(int)
```

✎✎✎

22.6 Μη–Αντιμεταθετικοί Δυαδικοί Τελεστές

Λέμε ότι ένας δυαδικός τελεστής είναι μη-αντιμεταθετικός όταν διαχείριζόμαστε με διαφορετικό τρόπο τα δύο μέρη του. Συνήθως αυτό συνοδεύεται και από διαφορά τύπου (τουλάχιστον κατά ένα "`const`").

22.6.1 Αντικείμενο Αριστερά

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν δυαδικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( Tl a, Tr b )`

  όπου Trv είναι ο τύπος του αποτελέσματος, Tl ο τύπος της πρώτης παραμέτρου και Tr ο τύπος της δεύτερης παραμέτρου.»

Τώρα συμπληρώνουμε: Αν έχουμε το αντικείμενο αριστερά, αν δηλαδή ο *Tl* είναι *C*& ή *C* ή `const` *C*& η επιφόρτωση γίνεται με μέθοδο της κλάσης *C* και πιο συγκεκριμένα:

- Αν ο *Tl* είναι *C*& με μέθοδο με μια παράμετρο:

  `Trv operator@( Tr b )`

  Αν ο *Tl* είναι *C* ή `const` *C*& με μέθοδο:

  `Trv operator@( Tr b ) const`

Ως παράδειγμα θα επιφορτώσουμε τον τελεστή "`[]`" για τη *BString* και τον "`+=`" για τη *BString* και για τη *Date*.

22.6.1.1 Ο Τελεστής "`[]`" για τη *BString*

Μετά τη δήλωση:

```
string q( "qazwsx" );
```

η εντολή:

```
cout << q << "  " << q[2] << endl;
```

δίνει:

```
qazwsx  z
```

Ακόμη οι:[2]

```
q[2] = 'a';
cout << q << endl;
```

δίνουν:

```
qaawsx
```

Για να είναι αυτό δυνατόν στη *BString*, ο τελεστής "`[]`" θα πρέπει να επιφορτώνεται ως:

```
char& operator[]( BString a, int b )
```

Πρόσεξε ότι επιστρέφει τιμή τύπου "`char&`" (τιμή-*l*) και όχι "`char`". Σύμφωνα με αυτά που είπαμε παραπάνω θα πρέπει να τον επιφορτώσουμε ως μέθοδο

```
char& operator[]( int pos ) const {  return bsData[pos];  }
```

Μετά από αυτό, αν έχουμε ορίσει:

```
BString q( "qazwsx" );
```

οι εντολές:

```
cout << q.c_str() << "  " << q[2] << endl;
q[2] = 'a';
cout << q.c_str() << endl;
```

θα δώσουν:

```
qaawsx
qaawsx  a
```

Το ίδιο αποτέλεσμα παίρνεις και με τις:

```
cout << q.c_str() << "  " << q.operator[](2) << endl;
q.operator[](2) = 'b';
cout << q.c_str() << endl;
```

22.6.1.2 Ο Τελεστής "+=" για τη *BString*

Έστω ότι θέλουμε να χρησιμοποιήσουμε τον τελεστή "`+=`" για να συνδέσουμε στο τέλος ορμαθού έναν άλλο ορμαθό.[3] Η επιφόρτωση θα είναι της μορφής:

```
BString& operator+=( BString& lhs, const BString& rhs )
```

ώστε στο πρόγραμμά μας να μπορούμε να γράφουμε:

```
s0 += s1;
```

Σύμφωνα με αυτά που είπαμε θα πρέπει να τον επιφορτώσουμε με μια μέθοδο:

```
BString& BString::operator+=( const BString& rhs )
```

---

[2] Να υπενθυμίσουμε: για την *std*`::`*string* η διαφορά των *at* και "`operator[]`" είναι ότι
- η *at* κάνει έλεγχο του δείκτη ενώ
- ο "`operator[]`" τον αφήνει στον προγραμματιστή.

[3] Αν s_1, s_2 ορμαθοί, τότε η s_1 `+=` s_2 ισοδυναμεί με $s_1 \leftarrow s_1 \frown s_2$, όπου "$\frown$" είναι ο τελεστής σύνδεσης ακολουθιών.

Το πρόγραμμα δεν είναι απλό. Ας δούμε τι περιπτώσεις υπάρχουν (είμαστε «μέσα στο *s0*» και *rhs* είναι το *s1*):

- Να μην χρειαστούμε επιπλέον μνήμη. Πότε συμβαίνει αυτό; Όταν *bsLen* + *rhs*.*bsLen* + 1 ≤ *bsReserved*. Στην περίπτωση αυτή όλα είναι απλά:

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

- Όταν *bsLen* + *rhs*.*bsLen* + 1 > *bsReserved* χρειαζόμαστε μνήμη.

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc )
      {  throw BStringXptn( "operator+=",
                            BStringXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις, αν δεν καταφέρουμε να πάρουμε μνήμη –και ριχτεί εξαίρεση– δεν θα αλλάξει η τιμή του αντικειμένου (ασφάλεια προς τις εξαιρέσεις επιπέδου ισχυρής εγγύησης).

Να και ένα παράδειγμα χρήσης:

```
      BString s2( "qazwsxedcrf" ), s3( "res" ), s4( "qwerty" );

      cout << s2.c_str() << "  " << s2.length() << endl
           << s3.c_str() << "  " << s3.length() << endl
           << s4.c_str() << "  " << s4.length() << endl;

      s4 += s3;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
      s4 += s2;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
```

Αποτέλεσμα:

```
qazwsxedcrf  11
res  3
qwerty  6
qwertyres  qazwsxedcrf  res
qwertyresqazwsxedcrf  qazwsxedcrf  res
```

Μην αμφιβάλλεις: αν γράψεις "`s4.operator+=(s3)`" αντί για "`s4 += s3`" και "`s4.operator+=(s2)`" αντί για "`s4 += s2`" θα πάρεις τα ίδια αποτελέσματα ακριβώς.

Στη *string* υπάρχει και η μέθοδος *append* που κάνει τα ίδια ακριβώς με τον "`+=`". Για αυτήν τη μέθοδο θα πρέπει να παραβιάσουμε τους κανόνες μας (σύμφωνα με τους οποίους θα πρέπει να την κάνουμε "`void`"): την υλοποιούμε αντιγράφοντας την υλοποίηση του τελεστή και βάζοντας μόνον στην επικεφαλίδα "`append`" όπου "`operator+=`":

```
BString& BString::append( const BString& rhs )
// ΤΑ ΥΠΟΛΟΙΠΑ ΙΔΙΑ ΜΕ ΤΟΝ operator+=
```

ή, ακόμη καλύτερα, με τον ορισμό (**inline**)

```
BString& append( const BString& rhs ) { return (*this += rhs); }
```

22.6.1.3 Ο Τελεστής "+=" για τη *Date*

Έστω ότι θέλουμε να γράψουμε μια μέθοδο για την κλάση *Date*, ας την πούμε *forword*, που θα προχωράει μια ημερομηνία. Δηλαδή, αν έχουμε δηλώσει:

```
Date d( 2007, 10, 15 );
```

τότε η:

```
d.forward( 5 );
```

θα προχωρήσει τη *d* κατά 5 ημέρες και η τιμή της θα γίνει 20.10.2007.

Σύμφωνα με τους κανόνες μας, θα πρέπει να τη δούμε ως "`forward(d, 5)`" όπου μεταβάλλεται η τιμή του πρώτου ορίσματος και να υλοποιήσουμε ως `void`.

Για την ίδια λειτουργία θα επιφορτώσουμε και τον τελεστή "`+=`", πράγμα πολύ φυσικό αφού το ίδιο κάνει και για τους ακέραιους. Εδώ όμως τα πράγματα αλλάζουν: Η "`d += 5`" θα πρέπει να είναι πράξη που θα επιστρέφει τη νέα τιμή της *d*. Επειδή γίνεται το ίδιο με τους αριθμούς; Ναι, αλλά και για τον εξής λόγο: έχουμε ήδη επιφορτώσει τον "`++`". Είναι επιθυμητό οι "`++d`" και "`d += 1`" να έχουν το ίδιο αποτέλεσμα. Η επιφόρτωση θα είναι της μορφής:

```
Date& operator+=( Date a, int dd )
```

Θα την υλοποιήσουμε ως μέθοδο

```
Date& operator+=( long int dd );
```

Η *forword()* θα πρέπει να συμπεριφέρεται παρομοίως. Θα έχουμε λοιπόν:

```
Date& forward( long int dd );
```

Ο πιο απλός τρόπος για να υλοποιήσουμε τις μεθόδους είναι να χρησιμοποιήσουμε τις συναρτήσεις διαχείρισης χρόνου της C. Στην επόμενη υποπαράγραφο θα δούμε εν συντομία αυτά που μας ενδιαφέρουν.

22.6.1.4 * Ο Χρόνος στη C

Για να χρησιμοποιήσεις τις συναρτήσεις διαχείρισης χρόνου σε ένα πρόγραμμα θα πρέπει να περιλάβεις το ctime:

```
#include <ctime>
```

Η πρώτη συνάρτηση που θα δούμε είναι η:

```
time_t time( time_t* timer );
```

που μας δίνει τα δευτερόλεπτα που έχουν περάσει από την 01.01.1970, ώρα 00:00:00 GMT[4]. Ο τύπος *time_t* είναι μετονομασία του `long int`.

Μπορείς να πάρεις τον τρέχοντα χρόνο στο πρόγραμμά σου δηλώνοντας:

```
time_t t1;
```

και δίνοντας: "`t1 = time(0)`" ή "`time(&t1)`".

Ελάχιστη επιτρεπόμενη τιμή της *t1* είναι το 0, που αντιστοιχεί σε 01.01.1970, ώρα 00:00:00 GMT (ή 02:00:00 ώρα Ελλάδας). Μέγιστη επιτρεπόμενη τιμή είναι η `LONG_MAX` που αντιστοιχεί σε 19.01.2038, ώρα 03:14:07 GMT (ή 05:14:07 ώρα Ελλάδας).

Η συνάρτηση

```
tm* localtime( const time_t* timer );
```

μετατρέπει μια τιμή τύπου `time_t` σε τιμή τύπου:

```
struct tm
{
   int tm_sec;     // seconds
   int tm_min;     // minutes
```

[4] GMT: Greenwich Mean Time. Η «ώρα Ελλάδος» είναι δύο ώρες μπροστά (02:00:00)

```
   int tm_hour;    // hour (0 - 23)
   int tm_mday;    // day of month (1 - 31)
   int tm_mon;     // month (0 - 11)
   int tm_year;    // year (Έτος - 1900)
   int tm_wday;    // μέρα εβδομάδας (0 – 6, 0 για Κυριακή)
   int tm_yday;    // μέρα του έτους (0 -365)
   int tm_isdst;   // για την αλλαγή θερινής ώρας.
}; // struct tm
```

Η αντίστροφη λειτουργία γίνεται από τη

```
time_t mktime( tm* t );
```

Πώς μπορούμε να χρησιμοποιήσουμε τα παραπάνω μαζί με τη *Date*; Ας πούμε ότι έχουμε μια τιμή:

```
   Date d1;
```

και θέλουμε να βάλουμε στην *t1* την τιμή τύπου **time_t** που ισοδυναμεί με *d1*. Δηλώνουμε μια:

```
   tm tm1;
```

και βάζουμε:

```
   tm1.tm_sec = 0;
   tm1.tm_min = 0;
   tm1.tm_hour = 12;   // μεσημέρι
   tm1.tm_mday = d1.dDay;
   tm1.tm_mon = d1.dMonth - 1;
   tm1.tm_year = d1.dYear - 1900;
   tm1.tm_wday = 0;
   tm1.tm_yday = 0;
   tm1.tm_isdst = 0;
```

Μετά χρησιμοποιούμε τη *mktime()*:

```
   t1 = mktime( &tm1 );
```

Αντιστρόφως, αν έχουμε την τιμή της *t1* παίρνουμε την αντίστοιχή της *d1* ως εξής:

```
   tm1 = *localtime( &t1 );
```

και στη συνέχεια:

```
   d1.dYear = tm1.tm_year + 1900;
   d2.dMonth = tm1.tm_month + 1;
   d2.dDay = tm1.tm_day;
```

22.6.1.5 * Υλοποίηση της *forward()*

Με βάση τα παραπάνω υλοποιούμε τη

```
   Date& Date::forward( int dd );
```

ως εξής:

- Δηλώνουμε τις:

```
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
```

- Μετατρέπουμε τις *dd* ημέρες σε 24×60×60×*dd* (= 86400×*dd*) δευτερόλεπτα:

```
   dd *= 86400;     // ημέρες σε δευτερόλεπτα
```

- Προσθέτουμε τη *dd* στην *currT*:

```
   currT += dd;
```

- Τέλος, μετατρέπουμε την τιμή *time_t* που προκύπτει στην «ισοδύναμη» τιμή *Date*:

```
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
```

Να ολόκληρη η μέθοδος:

```
Date& Date::forward( long int dd )
```

```
{
   const unsigned long secsPerDay = 86400;
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   dd *= secsPerDay;     // ημέρες σε δευτερόλεπτα
   if ( dd < 0 )
   {
      if ( currT < (-dd) )  // currT + dd < 0
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   else // dd >= 0
   {
      if ( dd > LONG_MAX - currT )  // currT + dd > LONG_MAX
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   currT += dd;
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
   return *this;
} // Date::forward
```

Στην **if** ελέγχουμε κατά πόσον η τιμή που προκύπτει είναι μέσα στα όρια.

Ο τελεστής "**+=**" υλοποιείται με τον ορισμό (**inline**)

```
   Date& operator+=( long int dd ) {  return forward( dd );  }
```

22.6.1.6 * Ο Τελεστής "++" της *Date* (ξανά)

Τώρα, για να είμαστε συνεπείς με αυτά που λέμε στην §22.2, θα πρέπει να ξαναορίσουμε την επιφόρτωση του "++" ως εξής:

```
   Date& operator++() {  return forward( 1 );  }
```

Καλό πράγμα η συνέπεια και η συμβατότητα, αλλά η προηγούμενη μορφή της επιφόρτωσης δεν ήταν ταχύτερη; Ναι, ήταν!

22.6.2 Ο Τελεστής "()" και η Χρήση του

Μια ιδιαίτερη περίπτωση δυαδικού τελεστή με το αντικείμενο αριστερά είναι ο τελεστής κλήσης συνάρτησης "**()**". Με την επιφόρτωση αυτού του τελεστή για μια κλάση C:

```
struct C
{
// . . .
   double operator()( double x ) const;
// . . .
};
```

αν έχεις δηλώσει

```
   C f;
```

μπορείς να γράφεις:

```
   if ( f(a)*f(b) < 0 ) m = f(a) + c/2;
// . . .
```

Δηλαδή, χρησιμοποιούμε το αντικείμενο *f* σαν συνάρτηση. Για τον λόγο αυτόν τέτοια αντικείμενα ονομάζονται **συναρτησιακά αντικείμενα** ή **συναρτησοειδή** (function objects, functors).

Όπως καταλαβαίνεις, από το παραπάνω παράδειγμα, αριστερά των παρενθέσεων βρίσκεται το όνομα του αντικειμένου αλλά στα δεξιά –και για την ακρίβεια μέσα στις παρεν-

θέσεις (όπως στην περίπτωση του "`[]`")– πρέπει να υπάρχει λίστα παραμέτρων που στο παράδειγμά μας είναι μονομελής αλλά, γενικώς, μπορεί να είναι πολυμελής ή και κενή.

Πού θα μπορούσε να είναι χρήσιμη αυτή η επιφόρτωση; Η πιο τυπική χρήση: στην αποφυγή της τεχνικής *συναρτήσεων ανάκλησης*. Ας δούμε ένα

Παράδειγμα

Ξαναγυρνούμε στη συνάρτηση *bisection()* στην §14.3. Η χρήση βέλους προς συνάρτηση δεν είναι και τόσο βολική. Να το αλλάξουμε λοιπόν, αλλά πώς;

Στο Παράδ. 2 της §14.7.1 λύσαμε ένα πρόβλημα που είχαμε με «μη βολικές» παραμετρους καταφεύγοντας σε παραμέτρους περιγράμματος. Μήπως μπορούμε να κάνουμε το ίδιο και εδώ; Δηλαδή: να κάνουμε

- τη *bisection()* περίγραμμα συνάρτησης και
- την *f* της "$f(x) = 0$" παράμετρο του περιγράμματος.

Καλό θα ήταν, αλλά *το περίγραμμα δεν δέχεται παράμετρο-συνάρτηση*! Δεχεται όμως παράμετρο κλάση και το πρόβλημά μας λύνεται αν κανονίσουμε να βάζουμε κλάση που τα αντικείμενά της είναι συναρτησοειδή. Γράφουμε λοιπόν:

```
template < class Func >
void bisection( double a, double b, double epsilon,
                double& root, int& errCode )
{
   Func   f;
   double m;

   if ( f(a)*f(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b - a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                            else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Όπως βλέπεις, ένα αντικείμενο κλάσης *Func*, όπως το *f*, θα πρέπει να μπορεί να χρησιμοποιηθεί ως συνάρτηση, για να έχουν νόημα τα ""`f(a)`", "`f(b)`", "`f(m)`".

Τώρα, αντί για συναρτήσεις θα γράψουμε κλάσεις:

```
class qF
{
public:
   double operator()( double x ) const
   {  return ( x - log(x) - 2 );  }
}; // qF


class cosF
{
public:
   double operator()( double x ) const
   {  return cos( x );  }
}; // cosF
```

Και να πώς θα γίνει το πρόγραμμα:

```
      bisection< qF >( 0.1, 1.0, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
      bisection< cosF >( 0.0, pi, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
```

```
                    else  cout << " Ρίζα = " << riza << endl;
```

Προσοχή: Ο μεταγλωττιστής θα δημιουργήσει δύο *bisection()* –δύο στιγμιότυπα του περιγράμματος–

- ένα για την εξίσωση $q(x) = 0$ και
- ένα για την εξίσωση $συν(x) = 0$.

Οι κλάσεις των συναρτησοειδών μπορεί να έχουν και αντίστοιχες κλάσεις εξαιρέσεων. Έτσι, μπορούμε να γράψουμε μια *qFXptn* και την *qF*· ακριβέστερα:

```
struct qFXptn
{
   enum { domain };
   char   funcName[100];
   int    errorCode;
   double errDblVal;
   qFXptn( const char* mn, int ec, double dv=0.0 )
      : errorCode( ec ), errDblVal( dv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // qFXptn

class qF
{
public:
   double operator()( double x ) const
   {
      if ( x <= 0 )
         throw qFXptn( "operator()", qFXptn::domain, x );
      return ( x - log(x) - 2 );
   }
}; // qF
```

Αν στο πρόγραμμά μας βάλουμε:

```
   try {
   // . . .
      bisection< qF >( -1, 1.0, 1e-5, riza, errCode );
   // . . .
   }
   catch( qFXptn& x )
   {
      cout << "domain error (" << x.errDblVal << ")" << endl;
   }
```

θα μας δώσει:

```
domain error (-1)
```

Η συνάρτηση

```
bool mult7( int a )  {  return ( (a%7) == 0 );  }
```

μας επιστρέφει **true** αν το όρισμα είναι διαιρετό δια 7, αλλιώς **false**. Αυτό είναι χαρακτηριστικό παράδειγμα **κατηγορήματος** (predicate).

Δες όμως και το πρόγραμμα

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

struct Mult7
{
   bool operator()( int a )  {  return ( (a%7) == 0 );  }
};

int main()
{
   Mult7 aMult7;

   cout << boolalpha;
```

```
   cout << aMult7(35) << "  " << aMult7(37) << endl;
}
```

που θα δώσει:

```
true  false
```

Και αυτό το συναρτησοειδές, που η συνάρτησή του είναι τύπου **bool**, είναι ένα *κατηγόρημα*.

22.6.2.1 * Μέλος – Περίγραμμα Συνάρτησης

Στην §21.12 είδαμε τα βέλη προς μεθόδους ως εργαλείο για τη χρήση της τεχνικής των συναρτήσεων ανάκλησης. Μήπως μπορούμε να χρησιμοποιήσουμε και στην περίπτωση αυτήν αυτά που μάθαμε παραπάνω για να απαλλαγούμε από τα βέλη προς μεθόδους; Για να κάνουμε κάτι τέτοιο θα πρέπει η *SList::toFile()* να γίνει περίγραμμα. Μπορεί να γίνει αυτό; Ναι, *η C++ σου επιτρέπει να βάλεις σε μια κλάση μέλη που να είναι περιγράμματα*. Έτσι στην περίπτωσή μας μπορείς να βάλεις:

```
class SList
{
public:
// . . .
   template < class OutFunc >
   void toFile( ostream& out ) const
   {
      OutFunc toStream;
      for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
         toStream( out, p->lnData );
   } // toFile
// . . .
};
```

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις τρεις κλάσεις, μια για κάθε μια από τις *save()*, *display()* και *writeToTable()* της *GrElmn*:

```
struct SaveGE
{
   void operator()( ostream& bout, const GrElmn& geObj )
   {  geObj.save( bout );  }
}; // SaveGE


struct DisplayGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.display( tout );  }
}; // DisplayGE


struct WriteToTableGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.writeToTable( tout );  }
}; // writeToTableGE
```

Όταν ο μεταγλωττιστής βρει στο πρόγραμμά μας την

```
      lst.toFile< DisplayGE >( tout );
```

θα εφοδιάσει αυτομάτως την *SList* με την αντίστοιχη μέθοδο.

22.6.3 Αντικείμενο Δεξιά

Εδώ αναφερόμαστε σε επιφορτώσεις της μορφής

```
Trv operator@( Tl a, Tr b )
```

Τέτοιοι τελεστές:

- Αν ο *Tl* είναι κλάση, μπορεί να επιφορτωθούν ως μέθοδοί της, όπως ήδη είδαμε.
- Αν ο *Tr* είναι κλάση, δεν είναι δυνατόν να επιφορτωθούν ως μέθοδοι αφού, όπως έχουμε πει, το αντικείμενο που μας ενδιαφέρει παίζει –στη μέθοδο– ρόλο πρώτης παραμέτρου. Μπορούμε να τους επιφορτώσουμε ως *καθολικές συναρτήσεις*, όπως ακριβώς μάθαμε στην §14.6.4.

Για παράδειγμα, στην §15.5 επιφορτώσαμε τον "`<<`" για τη *Date* ως:

```
ostream& operator<<( ostream& tout, const Date& rhs )
```

Αυτός:

- Θα μπορούσε να επιφορτωθεί ως μέθοδος στην *ostream*, αλλά
- Για τη *Date* θα επιφορτωθεί ως καθολική συνάρτηση.

Η επιφόρτωση που κάναμε στην §19.1.4 είναι μια χαρά:

```
ostream& operator<<( ostream& tout, const Date& rhs )
{
   return tout << rhs.getDay() << '.' << rhs.getMonth() << '.'
               << rhs.getYear();
} // operator<<( ostream& tout, const Date
```

Να τη δηλώσουμε `friend`; Δεν χρειάζεται, αφού οι τρεις "*get*" είναι `inline`.

Παρομοίως γίνεται και η επιφόρτωση του "`<<`" για τη *BString*:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   return tout << rhs.c_str();
} // operator<<( ofstream, BString
```

Αυτός είναι ένας απλός αλλά κάπως επιπόλαιος τρόπος. Η εκτύπωση μέσω της *c_str()* (και γενικώς η χρήση της) τελειώνει στον πρώτο `'\0'` που τυχόν θα βρεθεί. Στην §20.1 επικαλεστήκαμε ως λόγο επιλογής της συγκεκριμένης παράστασης για τα αντικείμενα *BString* το ότι μας επιτρέπει «*να έχουμε και χαρακτήρες* `'\0'` *στην τιμή που αποθηκεύουμε*.» Αν λοιπόν το *bsData* δείχνει κάτι σαν `"ab\0cd"` θα πρέπει η εκτύπωση να τελειώνει στο `'d'` και όχι στο `'b'`. Να πώς μπορεί να γίνει αυτό:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.bsLen; ++k )  tout << rhs.bsData[k];
   return tout;
} // operator<<( ofstream, BString
```

Αν τη γράψεις έτσι θα πρέπει να δηλώσεις και:

```
class BString
{
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
// . . .
```

Αφού όμως και η *length()* και ο `operator[]` είναι μέθοδοι `inline` μπορούμε να γράψουμε χωρίς επιπλέον κόστος:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

που δεν χρειάζεται να δηλωθεί ως `friend`.

22.7 Αντιμεταθετικοί Τελεστές

Ένας δυαδικός *αντιμεταθετικός τελεστές* είναι ένας τελεστής που δέχεται δύο ορίσματα και έχει την ίδια δράση αν αντιμεταθέσουμε το αριστερό με το δεξιό μέρος· για παράδειγμα ο "`+`", οι τελεστές σύγκρισης κλπ. Στην §22.1 είδαμε με ένα παράδειγμα γιατί δεν μπορούμε

να επιφορτώσουμε τον "==" για τη *BString* με μέθοδο. Παρόμοιο πρόβλημα υπάρχει και με τον "+" και άλλους παρόμοιους τελεστές για την ίδια κλάση (και όχι μόνο).

Οι τελεστές αυτοί υλοποιούνται με *καθολικές συναρτήσεις* που μπορεί να δηλωθούν **friend** για να γίνουν ταχύτερες.

Έτσι, στη *BString* θα κάνουμε την επιφόρτωση του "+" ως

```
   BString operator+( const BString lhs, const BString rhs );
```

Πώς; Αφού έχουμε επιφορτώσει τον "+=" υπάρχει μια πάγια τεχνική.

22.7.1 Από τον "@=" στον "@"

Έχοντας τον "+=" (γενικώς: τον "@=") παίρνουμε τον "+" (γενικώς: τον "@") ως εξής:

```
BString operator+( const BString& lhs, const BString& rhs )
{
   BString fv( lhs );
   fv += rhs;
   return fv;
} // BStribg operator+
```

Έτσι, έχουμε εξασφαλισμένη και τη συμβατότητα με τη λειτουργία του "+=".

Πρόσεξε τον τύπο του αποτελέσματος: δεν είναι *BString*& αλλά *BString*. Γιατί; Διότι η επιστρεφόμενη τιμή είναι τοπική μεταβλητή της συνάρτησης και η ζωή της τελειώνει όταν τελειώνει η εκτέλεση της συνάρτησης. Στον "+=" (και την *append*) επιστρεφόμενη τιμή είναι το ίδιο το αντικείμενο (`*this`) που επιζεί και μετά τον τερματισμό εκτέλεσης της μεθόδου.

Πρόσεξε ακόμη ότι η υλοποίηση δεν χρησιμοποιεί μέλη του αντικειμένου και επομένως δεν χρειάζεται να δηλωθεί ως `friend`.

Μπορούμε να χρησιμοποιήσουμε την ίδια τεχνική για να επιφορτώσουμε τον "+" για τη *Date*; Βεβαίως, αλλά πριν το κάνουμε θα πρέπει να σκεφτούμε και κάποια άλλα πράγματα.

Κατ' αρχήν πρέπει να συμφωνήσουμε στο νόημα της πράξης έτσι ώστε να είναι συμβατή με τον "+=". Τι σημαίνει συμβατή; Σημαίνει ότι οι: "`d = d + 5`" και "`d += 5`" ("`d.forwatd(5)`") θα έχουν το ίδιο αποτέλεσμα. Αυτό μας λέει ότι μπορούμε να χρησιμοποιήσουμε την παραπάνω τεχνική.

Το άλλο σημείο που πρέπει να προσέξουμε είναι η αντιμεταθετικότητα: μετά τη δήλωση:

```
   Date d( 2007, 10, 15 ), d1;
```

οι:

```
   d1 = d + 5;
```

και

```
   d1 = 5 + d;
```

θα πρέπει να έχουν το ίδιο αποτέλεσμα.

Δηλαδή, έχουμε να υλοποιήσουμε δύο τελεστές με το ίδιο σύμβολο:[5]

`+`: *Date* × `int` ⇸ *Date*

`+`: `int` × *Date* ⇸ *Date*

Πέρα από την αντιμεταθετικότητα του τελεστή έχουμε και μια *ασύμμετρία* λόγω της διαφοράς τύπου των δύο ορισμάτων. Έτσι, η επιφόρτωση του γίνεται όπως και στη *BString* αλλά είναι διπλή αφού έχουμε δύο συναρτήσεις:

```
Date operator+( const Date& lhs, int rhs )
{
   Date fv( lhs );
   fv += rhs;
   return fv;
} // Date operator+
```

[5] Γιατί μερικές συναρτήσεις; Δες την υλοποίηση του "+=" και θα καταλάβεις.

```
Date operator+( int lhs, const Date& rhs )
{
   return ( rhs + lhs );
} // Date operator+
```

Και εδώ, ο τύπος του αποτελέσματος δεν είναι *Date*& αλλά *Date*, για ίδιους λόγους που είπαμε και στη *BString*.

22.7.2 Σύγκριση Ημερομηνιών

Μια άλλη κατηγορία «συμμετρικών»[6] τελεστών που υλοποιούνται με καθολικές συναρτησεις (με δύο παραμέτρους) είναι οι τελεστές σύγκρισης. Έτσι, οι επιφορτώσεις των "`==`" και "`<`" για την κλάση *Date*, όπως τις κάναμε στην §19.1.4 είναι μια χαρά.

Τώρα θέλουμε να τονίσουμε μια καλή προγραμματιστική πρακτική που μπορεί να σε προφυλάξει από «δύσκολα» προγραμματιστικά λάθη:[7]

- ***Όταν επιφορτώνεις έναν τελεστή σύγκρισης "@" επιφόρτωσε και τον αντίθετό του με χρήση του "@".***

Αυτό σημαίνει ότι μετά τους "`==`" και "`<`" θα πρέπει να επιφορτώσουμε και τους "`!=`" και "`>=`" και μάλιστα ως εξής:

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

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

Παρατήρηση: ▶

Αν δηλώσεις τον δημιουργό της *Date* ως (§21.13.1):

```
   explicit Date( int yp, int mp = 1, int dp = 1 );
```

δεν μπορείς να γράψεις "`d == 2013`" ή "`2013 == d`" αλλά, υποχρεωτικώς, "`d == Date(2013)`" ή "`Date(2013) == d`" αντιστοίχως. Έτσι, δεν υπάρχει πια πρόβλημα αν επιφορτώσουμε τον "==" (και τους άλλους τελεστές σύγκρισης) με μεθόδους (με μια παράμετρο), π.χ.:

```
bool Date::operator==( const Date& rhs ) { /*. . .*/ }
```

Για τη *BString*, στον δημιουργό της οποίας δεν βάλαμε το "`explicit`", αυτό δεν ισχύει και η καθολική συνάρτηση είναι η μόνη σωστή επιλογή.◀

22.7.3 Οι Τελεστές Σύγκρισης της *BString*

Στη συνέχεια θέλουμε να προχωρήσουμε στην επιφόρτωση των τελεστών σύγκρισης ("`==`", "`<`", κλπ) στον τύπο *BString*. Θα πρέπει όμως πρώτα να συζητήσουμε τις συγκρίσεις ορμαθών χαρακτήρων, που θα χρειαστούμε στη συνέχεια.

Ας πούμε ότι έχουμε δύο πίνακες:

```
   char lhs[10], s2[12];
```

οι οποίοι περιέχουν κείμενα με μήκη *len1* και *len2*. Βάζουμε:

```
   minLen = min( len1, len2 );
```

---

[6] Γιατί «*συμμετρικών*» και όχι «*αντιμεταθετικών*»; Φυσικά, ο "`<`" δεν είναι αντιμεταθετικός: αν η *lhs* `<` *rhs* έχει τιμή `true` τότε η *rhs* `<` *lhs* έχει τιμή `false`. Εδώ με το *συμμετρικός* εννοούμε οτι από *συντακτική άποψη* ό,τι μπορεί να μπει στο ένα μέρος μπορεί να μπει και στο άλλο.

[7] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposites (such as "==" and "!="), it is appropriate to define both.*» Την πρωτοείδαμε στο Project 2 (§Prj02.3).

Αφού υπενθυμίσουμε ότι στις θέσεις *lhs*[*len*1] και *s*2[*len*2] υπάρχει ο φρουρός '\0', θεωρούμε τις:

```
k = 0;
while ( lhs[k] == s2[k] && k < minLen-1 ) ++k;
d = static_cast<int>(lhs[k]) - static_cast<int>(s2[k]);
```

Ας δούμε την τιμή που θα πάρει η *d* με μερικά παραδείγματα:

α) Έστω ότι στο *lhs* έχουμε **abcd** και στο *s*2 τα ίδια: **abcd**. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η **while** θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == *s*2[3] == '\0' η *d* θα πάρει τιμή 0.

β) Έστω ότι στο *lhs* έχουμε **abcd** και στο *s*2: **abce**. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η **while** θα σταματήσει και πάλι όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'e' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('e') = 100 – 101 = –1 < 0.

γ) Έστω ότι στο *lhs* έχουμε **abcd** και στο *s*2: **abcc**. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η **while** θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'c' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('c') = 100 – 99 = 1 > 0.

Αυτή είναι η βασική ιδέα:

- Αν το *lhs προηγείται αλφαβητικώς* (λεξικογραφικώς) *του s*2 (περίπτωση β) η *d* παίρνει *αρνητική* τιμή.
- Αν το *lhs έπεται αλφαβητικώς του s*2 (περίπτωση γ) η *d* παίρνει *θετική* τιμή.
- Αν το *lhs είναι ίσο με το s*2 (περίπτωση α) η *d* παίρνει τιμή 0 (μηδέν).

Η σύγκριση γίνεται, κατ' αρχήν, με βάση τη θέση των χαρακτήρων στον πίνακα χαρακτήρων του υπολογιστή. Το «κατ' αρχήν» θα το συζητήσουμε αργότερα.

Δες μερικά παραδείγματα ακόμη:

δ) Έστω ότι στο *lhs* έχουμε **ab** και στο *s*2: **abcd**. Θα έχουμε *minLen* = *min*(2, 4) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η **while** θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

ε) Έστω ότι στο *lhs* έχουμε **abcd** και στο *s*2: **ab**. Θα έχουμε *minLen* = *min*(4, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η **while** θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

στ) Έστω ότι στο *lhs* έχουμε **ab␣** και στο *s*2: **ab**. Θα έχουμε *minLen* = *min*(3, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η **while** θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[3] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

Εδώ έχουμε τρεις περιπτώσεις που παίρνουμε *d* == 0 χωρίς να έχουμε ισότητα. Τι σημαίνει όμως αυτό; Ότι όσο μπορούμε να συγκρίνουμε έχουμε ισότητες χαρακτήρων. Άρα, ο ορμαθός που τελειώνει μέχρι εκεί που μπορούμε να συγκρίνουμε προηγείται ενώ ο άλλος, με το μεγαλύτερο μήκος έπεται.

Με βάση τα παραπάνω, γράφουμε μια συνάρτηση-εργαλείο, εσωτερική της *BString*, για την υλοποίηση των μεθόδων που θα προδιαγράψουμε στη συνέχεια[8]:

```
int BString::stringCmpr( const char* lhs, const char* rhs, int n )
{
   int k( 0 );
   while ( lhs[k] == rhs[k] && k < n-1 ) ++k;
   return ( static_cast<int>(lhs[k]) - static_cast<int>(rhs[k]) );
```

[8] Η υλοποίηση που θα δώσουμε είναι μια παραλλαγή της υλοποίησης που δίνεται στην STL της Silicon Graphics Inc (SGI).

```
} // BString::stringCmpr
```

Η *stringCmpp()*[9] συγκρίνει τους πρώτους *n* (0 .. *n*–1) χαρακτήρες των *lhs* και *s*2 και επιστρέφει:

- *αρνητική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs προηγούνται αλφαβητικώς* (λεξικογραφικώς) των *n* πρώτων χαρακτήρων *του s*2,
- *θετική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs έπονται αλφαβητικώς* των *n* πρώτων χαρακτήρων *του s*2,
- 0 (μηδέν) αν οι *n* πρώτοι χαρακτήρες του *lhs είναι ίσοι* με τους αντίστοιχους *n* πρώτους χαρακτήρες *του s*2.

Πρόσεξε ότι δεν ελέγχουμε πουθενά την τιμή της *n*: αυτό θα πρέπει να το κάνει η κάθε μέθοδος που καλεί τη *stringCmpr()*.

Η *string* έχει τις εξής μεθόδους[10] για σύγκριση:

```
   int compare( const BString& rhs ) const;
   int compare( int pos, int n, const BString& rhs) const;
   int compare( int pos, int n,
                const BString& rhs, int pos2, int n2 ) const;
```

Όλες οι μορφές της *compare()* επιστρέφουν μιαν ακέραιη τιμή με την ίδια λογική που έχουμε για τη *stringCmpr()*. Με τη βοήθεια της *stringCmpr()* θα υλοποιήσουμε την πρώτη από αυτές τις μεθόδους για τη *BString*. Για τα παρακάτω παραδείγματα υποθέτουμε ότι έχουμε δηλώσει:

```
   BString bs1, bs2;
```

Η παράσταση `bs1.compare(bs2)` θα πρέπει να επιστρέφει τιμή *μικρότερη από, ίση με* ή *μεγαλύτερη από* 0 αν, αντιστοίχως, η τιμή της *bs*1 *προηγείται από, είναι ίση με* ή *έπεται της* τιμής της *bs*2. Μπορούμε λοιπόν να ορίσουμε:

```
int BString::compare( const BString& rhs ) const
{
   int fv( stringCmpr(bsData, rhs.bsData, min(bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = bsLen - rhs.bsLen;
   return fv;
} // BString::compare
```

Ας έλθουμε τώρα στους τελεστές. Θα επιφορτώσουμε τους τελεστές σύγκρισης με *καθολικές συναρτήσεις*. Μπορούμε να γράψουμε:

```
bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::stringCmpr(lhs.bsData, rhs.bsData,
                               min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
```

Φυσικά, όπως και στην περίπτωση του `operator<` της *Date*, θα πρέπει να δηλώσουμε μέσα στην κλάση ως `friend` τη συνάρτηση (τελεστή) `operator==`:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
public:
   BString( const char* cs = "" );
// ...
```

και ο ορισμός του τελεστή που δώσαμε πιο πάνω γίνεται νόμιμος.

---

[9] Η *stringCmpr()* είναι μια βοηθητική συνάρτηση, σαν τις *lastDay()* και *leapYear()* της *Date*.

[10] Δεν είναι ακριβώς έτσι. Αν ψάξεις λίγο το αρχείο string θα δεις ότι είναι πιο πολύπλοκα.

22.7.3.1 * Η Σειρά Ταξινόμησης

Οι συγκρίσεις που είδαμε παραπάνω στηρίζονται στη θέση των χαρακτήρων μέσα στο σύνολο χαρακτήρων του υπολογιστή. Αυτό δεν μας βολεύει καθόλου όταν θέλουμε να ταξινομήσουμε λέξεις κατά τη συνήθη αλφαβητική σειρά. Όπως είναι τοποθετημένα τα ελληνικά γράμματα στο πρότυπο ΕΛΟΤ 928 θα έχεις τα εξής περίεργα: το όνομα **ΝΙΚΟΛΑΪΔΗΣ** ταξινομείται μετά το **ΝΙΚΟΛΑΚΗΣ** και η λέξη **μικροψυχία** πριν από τα **μικροϋλικά**.

Αυτό το πρόβλημα λύνεται με τη **σειρά ταξινόμησης** (collating sequence). Αυτή είναι μια συνάρτηση που καθορίζει τη σειρά των χαρακτήρων για την ταξινόμηση. Για σύνολο 256 χαρακτήρων μπορείς να την υλοποιήσεις πολύ εύκολα με έναν πίνακα:

```
   int cs[256];
```

Σε κάθε στοιχείο του cs βάζουμε ως τιμή τη σειρά με την οποία θέλουμε να ταξινομείται. Π.χ.:

```
   for ( int k = 0; k <= 255; ++k )  cs[k] = k;
// ...
// Λατινικοί
   cs['A'] = cs['a'] = 65;
   cs['B'] = cs['b'] = 66;
// ...
// Ελληνικοί
   cs[256+'A'] = cs[256+'Ά'] = cs[256+'α'] = cs[256+'ά'] = 161;
   cs[256+'B'] = cs[256+'β'] = 162;
// ...
   cs[256+'Υ'] = cs[256+'Ύ'] = cs[256+'Ϋ'] = cs[256+'υ']
               = cs[256+'ύ'] = cs[256+'ϋ'] = cs[256+'ΰ'] = 213;
// ...
```

Θυμίσου ότι τα ελληνικά γράμματα, ως σταθερές τύπου (**signed**) **char**, έχουν αρνητική τιμή[11]. Αν δεν καταλαβαίνεις τι ακριβώς κάνουμε γύρισε πίσω στο Κεφ. 3.

Αφού η βάση του μηχανισμού σύγκρισης είναι η *stringCmpr()* την αλλάζουμε ως εξής:

```
int BString::stringCmprCS( const char* lhs, const char* rhs, int n ) const
{
   int k = 0;

   while ( cs[ lhs[k] ] == cs[ rhs[k] ] && k < n-1) ++k;
   return ( cs[ lhs[k] ] - cs[ rhs[k] ] );
} // BString::stringCmprCS
```

22.8 Τελεστές για τη *Vector3*

Επιστρέφουμε τώρα στην κλάση *Vector3*, που είδαμε στο Project 2, για να ξαναδούμε –όπως έχουμε υποσχεθεί– τις επιφορτώσεις τελεστών.

Εξ αρχής να πούμε το εξής: Η *Vector3* έχει τετριμμένη αναλλοίωτη, όλα τα μέλη είναι ανοικτά και έτσι δεν υπάρχει πρόβλημα για την επιφόρτωση όλων των τελεστών με καθολικές συναρτήσεις. *Αυτά που δώσαμε στο Project 2 είναι απολύτως σωστά*. Αυτά που θα πούμε εδώ είναι παραδείγματα εφαρμογής κάποιων από τις συνταγές που είδαμε στο παρόν κεφάλαιο.

Ας ξεκινήσουμε με τον (προθεματικό) *ενικό τελεστή* "-" (πρόσημο). Αν **v** = (*x*, *y*, *z*) τότε -**v** = (-*x*, -*y*, -*z*). Σύμφωνα με τη συνταγή της §14.6.4 θα έχουμε:

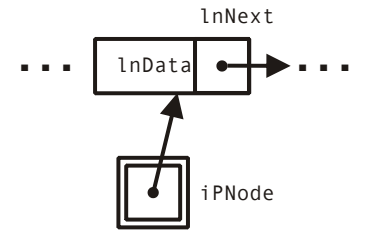
```
   Vector3 operator-( Vector3 rhs )
```

(ο τελεστής δεν αλλάζει το αντικείμενο) και σύμφωνα με τη συνταγή της §22.3.1 θα πρέπει να επιφορτώσουμε με μέθοδο:

```
   Vector3 operator-() const {  return Vector3( -x, -y, -z );  }
```

[11] Κανονικά πρέπει να γράψουμε `cs[256+static_cast<int>('A')] = 193` αλλά και έτσι περνάει.

Με μεθόδους θα πρέπει να επιφορτώσουμε και τους μη-αντιμεταθετικούς τελεστές με το αντικείμενο αριστερά "+=", "-=", "*=":



Σχ. 22-1 Ένας προσεγγιστής: το μοναδικό του μέλος, *iPNode*, είναι ένα βέλος που δείχνει κάποιον κόμβο της λίστας.

```
Vector3& Vector3::operator+=( const Vector3& rhs )
{
   x += rhs.x;  y += rhs.y;  z += rhs.z;
   return *this;
} // Vector3::operator+=

Vector3& Vector3::operator-=( const Vector3& rhs )
{
   x -= rhs.x;  y -= rhs.y;  z -= rhs.z;
   return *this;
} // Vector3::operator-=

Vector3& Vector3::operator*=( double rhs )
{
   x *= rhs;  y *= rhs;  z *= rhs;
   return *this;
} // operator*=( const Vector3&
```

Μέθοδοι θα γίνουν και οι *abs()* και *abs2()*:

```
   double abs2() const {  return x*x + y*y + z*z;  }
   double abs() const {  return sqrt( abs2() );  }
```

Οι υπόλοιπες καθολικές συναρτήσεις παραμένουν αλλά αλλάζουν οι ορισμοί των "`operator+`", "`operator-`", "`operator*`" που υλοποιούνται όπως μάθαμε στην §22.6.1, π.χ.:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv( lhs );
   fv += rhs;
   return fv;
} // operator+( const Vector3&
```

22.9 Διασχίζοντας τη Λίστα με τον "++" – Προσεγγιστές

Στην §21.11 λέγαμε: «*Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. ... η διάσχιση της λίστας δεν μπορεί να "κρυφτεί" σε κάποια μέθοδο.*» Έτσι, δίνουμε στο πρόγραμμα-πελάτη τη δυνατότητα να δίνει την εντολή "`p = p->lnNext`" με τον κίνδυνο να πάρει η *p->lnNext* τιμή που βγάζει από τη λίστα.

Τώρα ας εξετάσουμε την εξής ιδέα:

- όπως η "`++n`" προχωρεί την τιμή της *n* στον επόμενο ακέραιο,
- όπως η "`++d`" προχωρεί την τιμή της *d* στην επόμενη ημερομηνία

να επιφορτώσουμε τον "`++`" ώστε να προχωρεί τον *p* στον επόμενο κόμβο της λίστας. Και ακόμη να επιφορτώσουμε τον (ενικό) "`*`" ώστε η "`*p`" να μας δίνει το *p->lnData*. Έτσι δεν θα υπάρχει λόγος να «ανοίγουμε» τη δομή της λίστας στο πρόγραμμα-πελάτη.

Καλή ιδέα, αλλά έχει ένα προβληματάκι: η C++ δεν μας επιτρέπει να επιφορτώνουμε τελεστές για τους πρωτογενείς τύπους. Όμως αυτό το πρόβλημα λύνεται: Γράφουμε μια κλάση **προσεγγιστή** (iterator) που έχει ένα μόνον μέλος, το βέλος μας! Στο Σχ. 22-1 βλέπεις τι περίπου εννοούμε. Το βλέπεις και στη συνέχεια:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
```

```
public:
   class Iterator
   {
   public:
// . . .
      Iterator& operator++()
      {  iPNode = iPNode->lnNext;  return *this;  };
      GrElmn& operator*() {  return ( iPNode->lnData );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
// . . .
}; // SList
```

Εδώ όμως βλέπεις και *άλλα*:

- Η κλάση *SList* έχει τώρα δύο περιοχές **private**.
- Στην πρώτη περιοχή **private** έχουμε «κρύψει» τον ορισμό της *ListNode* που πρέπει να προηγείται της δήλωσης

```
      ListNode* iPNode;
```

- Στη νέα κλάση, που είναι κλάση «περιτυλίγματος», βλέπεις την επιφόρτωση των "`++`" και "`*`". Εδώ βρίσκεται το ενδιαφέρον!

Τι *άλλο* πρέπει να αλλάξουμε στην *SList*; Προφανώς τις *begin()* και *end()*· δεν είναι δυνατόν πια να επιστρέφουν βέλος. Θα τις κάνουμε:

```
   Iterator begin() {  return Iterator( slHead );  }
   Iterator end() {  return Iterator( slTail );  }
```

Προφανώς χρειάζεται να γράψουμε και έναν δημιουργό για τη νέα κλάση:

```
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
```

Πώς θα διασχίσουμε τη λίστα για να φυλάξουμε το περιεχόμενό της; Έτσι:

```
      for ( SList::Iterator it(lst.begin());
            it != lst.end();
            ++it )
         writeRandom( *it, bInOut );
```

Εδώ τα βλέπεις όλα:

- Χρησιμοποιούμε τον προσεγγιστή *it* για να πάμε από `lst.begin()` μέχρι `lst.end()`.
- Προχωρούμε με "`++it`".[12]
- Σε κάθε βήμα φυλάγουμε το περιεχόμενο ενός κόμβου που δίνεται από την αποπαραπομπή "`*it`".

Βλέπεις όμως ότι μας λείπει και κάτι: δεν επιφορτώσαμε τον "`!=`". Μπορούμε να τον επιφορτώσουμε με καθολική συνάρτηση:

```
bool operator!=( const SList::Iterator& a, const SList::Iterator& b )
{  return ( a.iPNode != b.iPNode );  }
```

αλλά θα πρέπει να την δηλώσουμε ως "`friend`".

Αφού έχουμε δηλώσει "`explicit Iterator`" δεν υπάρχει πρόβλημα αν κάνουμε την επιφόρτωση με μέθοδο:

---

[12] Οι προσεγγιστές που βλέπουμε εδώ έχουν μια διεύθυνση κίνησης: από την αρχή της λίστας προς το τέλος της («κινούνται» μόνον με τον "`++`"). Ένας τέτοιος προσεγγιστής καλείται **πρόσθιος** (forward) **προσεγγιστής**. Αν κάθε κόμβος είχε και βέλος προς τον προηγούμενο κόμβο θα ήταν δυνατή κίνηση του προσεγγιστή και από το τέλος προς την αρχή (και επιφόρτωση του "`--`"). Ένας τέτοιος προσεγγιστής λέγεται **αμφίδρομος** (bidirectional).

```
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
```

Κατά τη συνήθειά μας θα ορίσουμε και την:

```
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
```

Κλάση εξαιρέσεων θα γράψουμε; Ναι –κατ' αρχήν– διότι οι δύο βασικές επιφορτώσεις είναι ανοικτές σε λάθος χρήση. Δεν είναι όμως και λάθος να χρησιμοποιήσουμε την *SListXptn*.

- Ο "++" δεν μπορεί να χρησιμοποιηθεί αν έχουμε φτάσει στον φρουρό:

```
SList::Iterator& SList::Iterator::operator++()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator++", SListXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SList::Iterator::operator++
```

- Το ίδιο ισχύει και γα τον "*":

```
GrElmn& SList::Iterator::operator*()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator*", SListXptn::listEnd );
   return ( iPNode->lnData );
} // SList::Iterator::operator*
```

Ξαναδίνουμε το τμήμα του ορισμού της *SList* που έχει τις αλλαγές:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      GrElmn& operator*();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
   SList( const SList& rhs );
   ~SList();
   SList& operator=( const SList& rhs );
   void swap( SList& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
// . . .
}; // SList

struct SListXptn
{
   enum { noMemory, notFound, listEnd };
// . . .
```

```
}; // SListXptn
```

Παρατήρηση:▶

Τώρα που έχουμε τον προσεγγιστή και επιφορτώσαμε τους "++" και "*" –και έχοντας επιφορτώσει τον "==" για τη *GrElmn*– μπορούμε να χρησιμοποιήσουμε την (*std*::)*find()* για αναζήτηση στη λίστα.

Για να τη δοκιμάσεις βάλε στο πρόγραμμα της §21.11 τη νέα μορφή της *SList*, μην ξεχάσεις την "`include <algorithm>`" και στη **main**, μετά το τέλος της **do-while**, βάλε άλλη μια:

```
do {
   readAtNo( maxAtNo, aa );
   if ( aa != 0 )
   {
      SList::Iterator it;
      it = std::find( lst.begin(), lst.end(), GrElmn(aa) );
      if ( it != lst.end() )  (*it).display( cout );
                        else  cout << "not found" << endl;
   }
} while ( aa != 0 );
```

Σημείωσε τους ατομικούς αριθμούς των στοιχείων που θα βάλεις στη λίστα με την πρώτη **do-while** και σύγκρινε με αυτά που θα σου βγάζει η δεύτερη.◀

22.9.1 Επιφόρτωση του "->"

Αν επιφορτώσουμε τον "`->`" για την κλάση *SList*::*Iterator* μπορούμε αντί για

"`(*it).display(cout)`"

να γράψουμε

"`it->display(cout)`"

Ας δούμε πώς γίνεται αυτή η επιφόρτωση.

Ο "`->`" είναι, κατ' αρχήν, ένας δυαδικός τελεστής: αριστερά υπάρχει (για την περίπτωσή μας) ένας προσεγγιστής και δεξιά ένα μέλος ή μια μέθοδος της κλάσης που περιέχει η λίστα. Στη συνάρτηση που θα γράψουμε δεν υπάρχει δεύτερη παράμετρος (αντίστοιχη του δεξιού μέρους). Η συνάρτηση επιστρέφει τη διεύθυνση αυτού (βέλος προς αυτό) που επιστρέφει η `operator*()`. Και αφού η `operator*()` επιστρέφει "`iPNode->lnData`" θα έχουμε:

```
GrElmn* operator->() const
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator->",
                       SListXptn::listEnd );
   return ( &(iPNode->lnData) );
} // SList::Iterator::operator->
```

Η επιστρεφόμενη τιμή –που είναι βέλος– αντικαθιστά στην παράσταση τον προσεγγιστή και ο "`->`" δρα σε αυτό το βέλος. Στην περίπτωσή μας θα έχουμε:

`(&(iPNode->lnData))->display(cout)`

ή αλλιώς:

`(iPNode->lnData).display(cout)`

Αντί για "`it->display(cout)`" μπορείς να γράψεις:

"`(it.operator->())->display(cout)`"

22.10 * Απόκρυψη Υλοποίησης – Τεχνική "pimpl"

Και τώρα που τα μάθαμε όλα ας κλείσουμε μια εκκρεμότητα που έχουμε από την §19.3: να δούμε πώς αποκρύπτουμε την υλοποίηση της κλάσης (για κάποιον από τους λόγους που είδαμε στην §19.3.1).

Αυτό που θα θέλαμε να κάνουμε είναι το εξής: Να παραδίδουμε στον προγραμματιστή που θα χρησιμοποιήσει μια κλάση *C* (που έχουμε γράψει) το αρχείο **C.obj** (ή **C.o**) και το αρχείο **C.h** με τον ορισμό της κλάσης στον οποίον όμως θα υπάρχει μόνον το μέρος "**public**" με τις δηλώσεις των μεθόδων. Το μέρος "**private**" και οι ορισμοί των μεθόδων θα πρέπει να βρίσκονται μεταγλωττισμένα στο **C.obj**.

Ο συνδέτης δεν θα μας επιτρέψει κάτι τέτοιο. Θα μας επιτρέψει όμως κάτι που μοιάζει με αυτό: ένα μέρος "**private**" που θα έχει μέσα μόνο ένα βέλος. Ας δούμε αυτή τη λύση με ένα συγκεκριμένο παράδειγμα.

Η κλάση *BString*, όπως την έχουμε αναπτύξει μέχρι τώρα, είναι:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs ) { return (*this = rhs); }
   const char* c_str() const;
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   char& at( int k ) const;
   char& operator[]( int pos ) const {  return bsData[pos];  }
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs )
   { return (*this += rhs); }
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τώρα θα κάνουμε την εξής αλλαγή: Θα ορίσουμε μια άλλη κλάση:

```
struct BStringImpl
{
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BStringImpl
```

και στο μέρος "**private**" της κλάσης θα βάλουμε μόνο ένα βέλος:

```
private:
   BStringImpl* bsHandle;
}; // BString
```

Για να μπορέσουμε να κάνουμε αυτήν τη δήλωση για το *bsHandle* θα πρέπει ο μεταγλωττιστής να βρει *προειδοποιητική δήλωση* της *BStringImpl*:

```
   struct BStringImpl;
```

που λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

Μπορούμε να βάλουμε αυτήν τη δήλωση πριν από τον ορισμό της *BString* (στο **BString.h**). Είναι όμως προτιμότερο να τη βάλουμε μέσα στον ορισμό της κλάσης. Θα δεις στη συνέχεια γιατί αυτό μας συμφέρει.

Στο μέρος "**public**" δεν θα αφήσουμε οτιδήποτε έχει σχέση με υλοποίηση:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τα υπόλοιπα θα πάνε στο **BString.cpp**. Εκεί θα πάνε και ο ορισμοί της νέας κλάσης και των μεθόδων της.

Αν θέλεις μπορείς, όπως κάνουμε μέχρι τώρα, να βάλεις όσα αφορούν την κλάση *BStringImpl* σε χωριστά αρχεία **BStringImpl.h** και **BStringImpl.cpp**. Στην περίπτωση αυτή θα πρέπει να βάλεις "**#include "BStringImpl.h"**" και "**#include "BStringImpl.cpp"**" στην αρχή του **BString.cpp**.

Σημείωση: ▶
Τώρα μπορείς να καταλάβεις και το (συνηθισμένο) όνομα της τεχικής: ***p***ointer to (ή ***p***rivate) ***impl***ementation (βέλος προς την υλοποίηση). Θα συναντήσεις ακόμη τα ονόματα "handle class", "Cheshire Cat" και "compiler firewall". Η *BStringImpl* είναι μια **αδιαφανής** (opaque) δομή δεδομένων ενώ **αδιαφανές** ονομάζεται και το βέλος *bsHandle*· ονομάζεται ακόμη και **λαβή** (handle).

Λαβές για αδιαφανείς δομές δεδομένων χρησιμοποιούνται σε μικρή ή μεγάλη έκταση στις API για διάφορα προγραμματιστικά περιβάλλοντα.◀

Ας δούμε τώρα μερικά βασικά σημεία της υλοποίησης που θα αλλάξουν.

Ο ερήμην δημιουργός θα είναι:

```
BString::BString( const char* rhs )
{
   try {  bsHandle = new BStringImpl( rhs );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Αυτός ο δημιουργός μας επιβάλλει να γράψουμε έναν δημιουργό:

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

για την *BStringImpl*:

```
BString::BStringImpl::BStringImpl( const char* rhs )
{
   bsLen = cStrLen( rhs );
```

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

   bsData = new char[bsReserved];
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BStringImpl::BStringImpl
```

Πρόσεξε ότι η νέα κλάση αναφέρεται ως "`BString::BStringImpl`".

Αν αποτύχει η "`new`" και ριχτεί *bad_alloc* θα συλληφθεί από την "`catch`" του δημιουργού της *BString*. Και αν σε κάποιο πρόγραμμα έχουμε τη δήλωση:

```
   BStringImpl a;
```

και κατά την εκτέλεση του ερήμην δημιουργού ριχτεί μια *bad_alloc* πού θα συλληφθεί; Το ότι η κλάση μας είναι δηλωμένη στο μέρος "`private`" της *BString* μας εξασφαλίζει ότι δεν θα υπάρξει τέτοια δήλωση· θα την «απαγορεύσει» ο μεταγλωττιστής.

Αφού κάθε αντικείμενο δεσμεύει δυναμική μνήμη –για ένα αντικείμενο *BStringImpl*– θα πρέπει να γράψουμε δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
   try {  bsHandle = new BStringImpl( *(rhs.bsHandle) );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Στη "`new BStringImpl( *(rhs.bsHandle) )`" καλείται ο δημιουργός αντιγραφής της *BStringImpl*. Αφού ένα αντικείμενο αυτής της κλάσης δεσμεύει επίσης μνήμη για το *bsData* θα πρέπει να γράψουμε δημιουργό αντιγραφής και για αυτήν:

```
BString::BStringImpl::BStringImpl( const BString::BStringImpl& rhs )
{
   bsData = new char[rhs.bsReserved];
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BStringImpl::BStringImpl
```

Αυτά που είπαμε για τις *bad_alloc* από τους ερήμην δημιουργούς των δύο κλάσεων ισχύουν και για τους δημιουργούς αντιγραφής.

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε (επιφορτώσουμε) και τελεστή εκχώρησης:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
} // BString::operator=
```

Αυτός είναι ολόιδιος με αυτόν που γράψαμε στην §21.6.1 (τελευταία μορφή) αλλά έχουμε διαφορετική *swap()*:

```
void BString::swap( BString& rhs )
{
   std::swap( bsHandle, rhs.bsHandle );
} // BString::swap
```

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε τελεστή εκχώρησης και για τη *BStringImpl*:

```
BString::BStringImpl& BString::BStringImpl::operator=(
                                          const BString::BStringImpl& rhs )
```

```
{
   BString::BStringImpl tmp( rhs );
   if ( &rhs != this )
      swap( tmp );
   return *this;
} // BString::BStringImpl::BStringImpl& operator=
```

Βέβαια, προς το παρόν μας είναι άχρηστος αλλά μπορεί να μας χρειαστεί αν συνεχίσουμε την ανάπτυξη της *BString*.

Οι καταστροφείς είναι:

```
BString::~BString() {  delete bsHandle;  }
```

και

```
BString::BStringImpl::~BStringImpl() {  delete bsData;  }
```

Άλλες μεθόδους για τη *BStringImpl* θα γράψουμε; Μόνον τη *swap()*:

```
void BString::BStringImpl::swap( BString::BStringImpl& rhs )
{
   std::swap( bsData, rhs.bsData );
   std::swap( bsLen, rhs.bsLen );
   std::swap( bsReserved, rhs.bsReserved );
} // BString::BStringImpl::swap
```

Όπως βλέπεις πρόκειται για την παλιά `BString::swap` σε νέα συσκευασία.

Οι μέθοδοι της *BString* πρέπει να ξαναγραφούν με τις εξής τροποποιήσεις: Θα πρέπει να γράφεις

- το πρόθεμα "`bsHandle->`" πριν από τα *bsData, bsLen, bsReserved*.
- το πρόθεμα "`BStringImpl::`" πριν από τη σταθερά *bsIncr* και τις (στατικές) βοηθητικές συναρτήσεις *cStrLen, stringCmpr*.

Ας δούμε πώς θα γραφεί η επιφόρτωση του "`+=`":

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 >
                                          bsHandle->bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsHandle->bsLen+(rhs.bsHandle)->bsLen+1)/
                     BStringImpl::bsIncr+1)*BStringImpl::bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "operator+=",
                            BStringXptn::allocFailed );  }
      for ( int k(0); k < bsHandle->bsLen; ++k )
         tmp[k] = bsHandle->bsData[k];
      delete[] bsHandle->bsData;
      bsHandle->bsData = tmp;
   }
   for ( int j(0), k(bsHandle->bsLen);
         j < (rhs.bsHandle)->bsLen;
         ++j, ++k )
      bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];
   bsHandle->bsLen += (rhs.bsHandle)->bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις:

- Στη συνθήκη της **if** αντί για
  `bsLen + rhs.bsLen + 1 > bsReserved`
  γράφουμε
  `bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 > bsHandle->bsReserved`
- Στην αρχική τιμή της *tmpRes* γράφουμε "`BStringImpl::bsIncr`" και όχι `bsIncr`.
- Στη δεύτερη **for** αντιγράφουμε με τη
  `bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];`

και όχι με τη

```
bsData[k] = rhs.bsData[j];
```

Τα ίδια ισχύουν και για τις επιφορτώσεις των τελεστών "==" και "<<" που τώρα τις έχουμε δηλώσει "**friend**":[13]

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < (rhs.bsHandle)->bsLen; ++k )
      tout << (rhs.bsHandle)->bsData[k];
   return tout;
} // operator<<( ofstream, BString

bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::BStringImpl::stringCmpr(
                                          (lhs.bsHandle)->bsData,
                                          (rhs.bsHandle)->bsData,
             min((lhs.bsHandle)->bsLen, (rhs.bsHandle)->bsLen)) );
   if ( fv == 0 )
      fv = (lhs.bsHandle)->bsLen - (rhs.bsHandle)->bsLen;
   return ( fv == 0 );
} // operator==
```

Ακολουθώντας αυτά που είπαμε στην §18.3 μπορούμε από το **BString.cpp** (και το **BString.h**) να πάρουμε το μεταγλωττισμένο αρχείο **BString.o** (ή **BString.obj**).

Τώρα το **BString.h** είναι:

```
#ifndef _BSTRING_H
#define _BSTRING_H

#include <fstream>
#include <string>

using namespace std;

class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );

struct BStringXptn
{
   enum { allocFailed, outOfRange };
```

[13] Η επιφόρτωση του "+" παραμένει όπως δόθηκε στην §22.6.1.

```
   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

#endif // _BSTRING_H
```

Ένας προγραμματιστής, έχοντας το **BString.o** (ή **.obj**) και το **BString.h**, *μπορεί να χρησιμοποιήσει στα προγράμματά του τη BString χωρίς να ξέρει πώς είναι γραμμένη.*

Και ακόμη: αν εμείς αλλάξουμε την υλοποίηση της κλάσης χωρίς να αλλάξουμε τη διεπαφή (δηλαδή αυτά που έχουμε στο "**public**" και τις επικεφαλίδες των καθολικών συναρτήσεων) οποιοδήποτε πρόγραμμα χρησιμοποιεί την κλάση δεν χρειάζεται να περάσει ξανά από τον μεταγλωττιστή· αρκεί να περάσει από τον συνδέτη.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

22-1 Επιφόρτωσε τους τελεστές "**-**", "**--**" και "**-=**" για τη *Date* ώστε να δίνουν αντιστοίχως τη διαφορά (σε ημέρες) δύο ημερομηνιών και την «οπισθοδρόμηση» μιας ημερομηνίας.

22-2 Επιφόρτωσε όλους τους τελεστές σύγκρισης για τις *Date* και *BString*.

project

4

Φοιτητές και Μαθήματα Αλλιώς

Περιεχόμενα:

Prj04.1 Το Πρόβλημα 771
Prj04.2 Η Κλάση *Course* 772
Prj04.3 ... Και ο «Πίνακας Μαθημάτων» 779
Prj04.3.1 Οι Μέθοδοι *add1Course()* και *delete1Course()* 783
Prj04.3.2 Οι *save()* και *load()* 785
Prj04.3.3 Απώλειες Πρόσβασης 786
Prj04.3.4 Επιφορτώνουμε τον " [ ] "; 787
Prj04.4 Περί Διαγραφών 788
Prj04.5 Η Κλάση *Student* 788
Prj04.5.1 Ο «Κανόνας των Τριών» 789
Prj04.5.2 Μέθοδοι "*get*" και "*set*" 790
Prj04.5.3 Μέθοδοι για τα Στοιχεία του Πίνακα 791
Prj04.5.4 Φύλαξη και Φόρτωση 793
Prj04.5.5 Η Κλάση *Student* 794
Prj04.6 Το «Μητρώο Φοιτητών» 795
Prj04.6.1 Φύλαξη, Φόρτωση και Ευρετήριο 798
Prj04.7 Η Κλάση *StudentInCourse* 799
Prj04.8 Η Κλάση *StudentInCourseCollection* 802
Prj04.9 Πώς θα Γίνονται οι Ενημερώσεις 806
Prj04.10 Οι Άλλες Συλλογές Τελικώς 807
Prj04.10.1 Η Κλάση *CourseCollection* 807
Prj04.10.2 Η Κλάση *StudentCollection* 810
Prj04.11 Το 1ο Πρόγραμμα – Δημιουργία 812
Prj04.11.1 Αρχείο Φοιτητών και Δηλώσεων Μαθημάτων 813
Prj04.11.2 Έλεγχος Δηλώσεων 815
Prj04.11.3 Φύλαξη 816
Prj04.11.4 ...Και το Πρόγραμμα 816
Prj04.12 Το 2ο Πρόγραμμα – Εκμετάλλευση 818
Prj04.13 Για το Παράδειγμά μας 821

Prj04.1 Το Πρόβλημα

Το πρόβλημα που έχουμε να λύσουμε είναι μια παραλλαγή του προβλήματος που λύσαμε στο Project 3. Οι επιπλέον απαιτήσεις είναι:

Α. Σε κάθε αντικείμενο κλάσης Student θα αποθηκεύονται –σε δυναμικό πίνακα– οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.

Αυτό θα μας δημιουργήσει το εξής πρόβλημα: Όταν φυλάγουμε τα αντικείμενα τύπου *Student* στο `students.dta` αυτά θα έχουν διαφορετικό μέγεθος. Έτσι, δεν θα μπορούμε να χειριστούμε το αρχείο με τον τρόπο που μάθαμε στην §15.13. Για να ανακτήσουμε τη δυνα-

τοτητα κατ' ευθείαν πρόσβασης στα αντικείμενα του αρχείου θα χρειαστούμε ένα **ευρετήριο** (index) με στοιχεία τύπου

```
struct IndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
}; // IndexEntry
```

Το πρόγραμμά μας θα πρέπει να οργανώνει αυτά τα στοιχεία σε έναν πίνακα index *που θα τον φυλάγει σε ένα (μη-μορφοποιημένο) αρχείο* `students.ndx`.

Β. Εκτός από το πρόγραμμα που γράψαμε –και θα ξαναγράψουμε με κάποιες διαφορές– και το ονομάζουμε ***πρόγραμμα δημιουργίας****, θέλουμε άλλο ένα πρόγραμμα που θα το ονομάσουμε* ***πρόγραμμα εκμετάλλευσης****.*

Αυτό το πρόγραμμα θα κάνει κατά βάση μια δουλειά: θα δέχεται από τον χρήστη έναν αριθμό μητρώου φοιτητή/τριας και θα δείχνει τα στοιχεία του/της όπως υπάρχουν στο `students.dta`*. Αν δεν υπάρχει τέτοιος αριθμό μητρώου θα δίνει κατάλληλο μήνυμα. Αυτό θα επαναλαμβάνεται μέχρι να ζητηθεί "ΤΕΛΟΣ" από τον χρήστη.*

Ακόμη, αυτή τη φορά:

- θα γράψουμε τις κλάσεις μας με βάση τη «συνταγή» της §21.8,
- θα χειριστούμε με πιο «ορθόδοξο» τρόπο τους εξωτερικούς πίνακες μαθημάτων και φοιτητών και
- θα γράψουμε κάπως διαφορετικά τις κλάσεις εξαιρέσεων: κάθε αντικείμενο-εξαίρεση θα περιέχει και το κλειδί του αντικειμένου που έχει το πρόβλημα. Το είχαμε υποσχεθεί στην §19.4.

Σε σχέση με το πρώτο σημείο θα πρέπει να τονίσουμε το εξής: *Ο σωστός τρόπος εργασίας είναι αυτός που είδαμε στο Project 3, δηλαδή βλέπουμε τις ανάγκες για την κάθε κλάση όπως υπαγορεύονται από τις εφαρμογές που τη χρησιμοποιούν και την εξοπλίζουμε καταλλήλως.* Η «συνταγή» της §21.8 ασχολείται με μερικές πολύ συνηθισμένες ανάγκες που καλό είναι να αντιμετωπίζονται με πάγιο τρόπο.

Prj04.2 Η Κλάση *Course*

Πριν αρχίσουμε να εφαρμόζουμε τη «συνταγή» μας για την κλάση *Course* θα πρέπει να πάρουμε υπόψη μας κάτι που υπάρχει στις προδιαγραφές και αφορα την κλάση *Student*: «*Σε κάθε αντικείμενο κλάσης Student θα αποθηκεύονται –σε δυναμικό πίνακα– οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.*»

Είναι φανερό ότι στον πίνακα αυτόν θα πρέπει να κάνουμε αναζητήσεις. Αν θέλουμε να χρησιμοποιήσουμε το περίγραμμα *linSearch()* θα πρέπει να έχουμε επιφορτώσει τον τελεστή "`!=`" για τον τύπο του *cCode*. Αυτό δεν μπορεί να γίνει αν έχουμε δηλώσει:

```
   char       cCode[cCodeSz];
```

Θα πρέπει να ορίσουμε έναν νέο τύπο (κλάση περιτυλίγματος), ας τον πούμε *CourseKey*, ως εξής:

```
class Course
{
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strcpy( s, aKey.c_str(), cCodeSz-1 );  s[cCodeSz-1] = '\0';  }
   }; // CourseKey
// . . .
```

και να δηλώσουμε:

```
// . . .
private:
// . . .
   CourseKey    cCode;            // κωδικός μαθήματος
```

Τώρα μπορούμε να επιφορτώσουμε τον "`!=`" ως εξής:

```
bool operator!=( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) != 0 );  }
```

Και τώρα ας εφαρμόσουμε τη συνταγή μας.

ΑΝ, αναλλοίωτη:

((*strlen*(*cCode*.*s*) == *cCodeSz*-1) || (*strlen*(*cCode*.*s*) == 0)) && (*strlen*(*cTitle*) < *cTitleSz*) &&
(1 <= *cFSem* <= 8) && (*cSector* ∈ {'Υ','Π','Μ','Γ','Ξ'})[1] &&
(*cCateg* ∈ {"ΜΓΥ","ΜΕΥ","ΜΕ","ΔΟΝ"})[2] && (*cWH* >= 0) && (*cUnits* >= 0) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) || (*strlen*(*cPrereq*.*s*) == 0)) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) ⇒ (*cPrereq* != *cCode*)) &&
(*cNoOfStudents* >= 0)

Παρατηρήσεις: ▶

1. Οι *strlen*(*cCode*.*s*) == *cCodeSz*-1 και *strlen*(*cTitle*) < *cTitleSz* είναι διαφορετικές: Ο κωδικός του μαθήματος πρέπει να έχει ακριβώς 7 χαρακτήρες (και άλλα χαρακτηριστικά δομής –που περιγράφονται στην άσκ. Prj03-1– αλλά εδώ τα αγνοούμε.) *Αν δοθούν λιγότεροι ή περισσότεροι θα πρέπει να ρίξουμε εξαίρεση*. Ο τίτλος του μαθήματος πρέπει να έχει λιγότερους από 80 χαρακτήρες. Αν μας δοθούν 80 ή περισσότεροι *δεν ρίχνουμε εξαίρεση*· απλώς κρατούμε τους πρώτους 79 (*cTitleSz*-1) και αγνοούμε τους παραπανήσιους.
2. Η *strlen*(*cCode*.*s*) == 0 θα πρέπει να επιτρέπεται στην περίπτωση δήλωσης της μορφής "`Course c0`" (και μόνον).
3. Οι *cWH* == 0 και *cUnits* == 0 είναι δεκτές μόνον στον δημιουργό και όχι στις αντίστοιχες "*set*".
4. Η

((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) || (*strlen*(*cPrereq*.*s*) == 0)) &&
((*strlen*(*cPrereq*.*s*) == *cCodeSz*-1) ⇒ (*cPrereq* != *cCode*))

δεν φτάνει: θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων. Για να μπορέσεις να βάλεις αυτόν τον περιορισμό στην αναλλοίωτη και τον αντίστοιχο έλεγχο στις μεθόδους θα πρέπει να σχεδιάσεις τις κλάσεις σου έτσι ώστε τα αντικείμενα τύπου *Course* να υπάρχουν μόνον μέσα σε κάποιον «πίνακα μαθημάτων».
5. Ο έλεγχος «αυτοαναφοράς» (κωδικός προαπαιτουμένου ίδιος με κωδικό μαθήματος) θα πρέπει να γίνεται όταν αλλάζουμε κωδικό προαπαιτουμένου αλλά και όταν αλλάζουμε κωδικό μαθήματος.◀

Η κλάση εξαιρέσεων θα είναι ως εξής:

```
struct CourseXptn
{
   enum { . . . };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
               const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
```

[1] Έχουν σχέση με το Τμήμα Βιομηχανικής Πληροφορικής του ΤΕΙ Καβάλας.

[2] Αναφέρονται στα προγράμματα σπουδών ΤΕΙ κάποιας εποχής...

```
}; // CourseXptn
```

Το νέο στοιχείο εδώ είναι το μέλος *objKey*. Εδώ θα βάζουμε το κλειδί του αντικειμένου που έχει το πρόβλημα.

Να σημειώσουμε ακόμη ότι για να μπορούμε να χρησιμοποιήσουμε στη δήλωση του *objKey* τη σταθερά *Course*`::`*cCodeSz* θα πρέπει να μεταφέρουμε τον ορισμό της σε περιοχή "`public`" στη δήλωση της *Course*.

ΔΕ, ερήμην δημιουργός: Θέλουμε να έχουμε δυνατότητα να κάνουμε δηλώσεις σαν τις:

```
   Course c0;
   Course c1( "EY01010" );
   Course c2( "EY01010", "Εισαγωγή στον Προγραμματισμό (Θ)" );
```

Σύμφωνα με αυτά που είπαμε στην §21.11.1, επειδή –όπως φαίνεται στο δεύτερο από τα παραδείγματα– ο δημιουργός θα επιτρέπει και τη μετατροπή τύπου (κλήση με μια και μόνη παράμετρο τύπου *string*), δηλώνουμε:

```
   explicit Course( string aCode="", string aTitle="" );
```

και ορίζουμε:

```
Course::Course( string aCode, string aTitle )
{
   if ( aCode.length() != cCodeSz-1 &&
        (aCode.length() != 0 || aTitle.length() != 0) )
      throw CourseXptn( CourseKey(""), "Course", CourseXptn::keyLen,
                        aCode.c_str() );
   cCode = CourseKey( aCode );
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
   cFSem = 0;
   cCompuls = false;
   cCateg[0] = '\0';
   cWH = 0;
   cUnits = 0;
   cPrereq = CourseKey( "" );
   cNoOfStudents = 0;
} // Course::Course
```

Να εξηγήσουμε τον έλεγχο που κάνουμε: Ο κωδικός μαθήματος θα πρέπει να έχει μήκος 7. Μπορεί να έχει και μήκος 0 αλλά μόνον σε δήλωση της μορφής "`Course c0`". Όμως δεν επιτρέπεται δήλωση της μορφής:

```
   Course c2( "", "Εισαγωγή στον Προγραμματισμό (Θ)" );
```

όπου έχουμε τίτλο μαθήματος αλλά όχι κωδικό. Δηλαδή, πιο σωστά, θα έπρεπε να έχουμε στην αναλλοίωτη:

(*strlen*(*cCode*`.`*s*) == *cCodeSz*-1) `||` (*strlen*(*cCode*`.`*s*) == 0 && *strlen*(*cTitle*) == 0)

Η συνθήκη ελέγχου είναι άρνηση αυτής της συνθήκης για τις αντίστοιχες παραμέτρους του δημιουργού.

Αν μας δώσουν «παράνομο» κωδικό μαθήματος στον δημιουργό τι κλειδί θα βάλουμε στην εξαίρεση; Όπως βλεπεις, εδώ βάλαμε "`CourseKey("")`" (κενός ορμαθός). Η «αμαρτωλή» τιμή (*aCode*) περνάει με το τελευταίο όρισμα.

Κατά τα άλλα, αντί για "`cCode = CourseKey( aCode )`" θα μπορούσαμε να είχαμε γράψει:

```
   CourseKey cCode;
   strcpy( cCode.s, aCode.c_str() );
```

και αντί για "`cPrereq = CourseKey( "" )`" θα μπορούσαμε να έχουμε:

```
   cPrereq.s[0] = '\0';
```

Παρατήρηση: ▶

Μήπως έπρεπε να βάλουμε το "**explicit**" και στον δημιουργό της *CourseKey*; Όχι! Η τιμή που μας ενδιαφέρει είναι ακριβώς τύπου "`char[cCodeSz]`". Την «μεταμφιέσαμε» σε "**struct**" για τεχικούς λόγους.◀

ΔΑ, δημιουργός αντιγραφής: Τα αντικείμενα της κλάσης δεν δεσμεύουν *δυναμικώς* πόρους του συστήματος. Έτσι, δεν απαιτείται να γράψουμε δημιουργό αντιγραφής· αυτός που γράφεται αυτομάτως από τον μεταγλωττιστή είναι σωστός.
ΤΕ, τελεστής εκχώρησης: Για τον ίδιο λόγο δεν χρειάζεται να γράψουμε και τελεστή εκχώρησης.
ΚΑ, καταστροφέας: Ούτε καταστροφέα χρειάζεται να γράψουμε· εμείς όμως θα βάλουμε έναν "κενό":

```
   ~Course() { };
```

GE, συναρτήσεις "*get*":

```
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   const char* getPrereq() const {  return cPrereq.s;  }
```

Πρόσεξε την πρώτη και την τελευταία: Προτιμούμε να επιστρέψουμε τιμή τύπου "`const char*`" –δηλαδή τον «φυσικό» τύπο των μελών– και όχι `CourseKey`, που είναι «μεταμφιεσμένος».
SE, συναρτήσεις "*set*": Ξεκινούμε με τη *setCode* που τώρα έχει και ελέγχους:

```
void Course::setCode( string aCode )
{
   if ( aCode.length() != cCodeSz-1 )
      throw CourseXptn( CourseKey(cCode), "setCode", CourseXptn::keyLen,
                        aCode.c_str() );
   if ( (aCode.length() != 0) && (CourseKey(aCode) == cPrereq) )
      throw CourseXptn( CourseKey(cCode), "setCode", CourseXptn::autoRef,
                        aCode.c_str() );
   strcpy( cCode.s, aCode.c_str() );
} // Course::setCode
```

Πρόσεξε ότι

- Τώρα δεν δεχόμαστε κωδικούς που το μήκος τους δεν είναι 7.[3]
- Για να γίνει η σύγκριση "`CourseKey(aCode) == cPrereq`" θα πρέπει να έχουμε επιφορτώσει τον "`==`" για τον *CourseKey*:

```
bool operator==( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( !(lhs != rhs) );  }
```

Είναι λάθος να γράψουμε "`!(CourseKey(aCode) != cPrereq)`" αφού επιφορτώσαμε ήδη τον "`!=`"; Μια χαρά είναι! Απλώς, αυτό που γράφουμε με τον "`==`" είναι πιο κομψό.

Σημείωση:▶
Πρόσεξε τον τρόπο επιφόρτωσης του "`==`": γίνεται με βάση τον "`!=`". Δεν θα ήταν πιο γρήγορος αν γράφαμε "`return ( strcmp(lhs.s, rhs.s) == 0 )`"; Ναι, αλλά έτσι που το γράψαμε έχουμε σίγουρη συμβατότητα των δύο τελεστών. Αν πάμε για τα πιο γρήγορα μπορεί να κάνουμε κάποιο λάθος, μπορεί, αν χρειαστεί να κάνουμε κάποια αλλαγή, να ξεχάσουμε να την κάνουμε και στις δύο συναρτήσεις και άλλα τέτοια.◀

Η *setTitle()* είναι απλή:

```
void Course::setTitle( string aTitle )
```

---

[3] Δηλαδή αν θέλουμε να «καθαρίσουμε» ολόκληρο το αντικείμενο δεν θα μπορούμε να «καθαρίσουμε» τον κωδικό; Στην περίπτωση αυτή δεν καθαρίζουμε τα μέλη ένα προς ένα με τις "*set*"· χρησιμοποιούμε τον (ερήμην) δημιουργό χωρίς ορίσματα:

```
   c0 = Course();
```

```
{
   strncpy( cTitle, aTitle.c_str(), cTitleSz-1 );  cTitle[cTitleSz-1] = '\0';
} // Course::setTitle
```

Η

```
void Course::setFSem( int aFSem )
{
   if ( aFSem < 1 || 8 < aFSem)
      throw CourseXptn( cCode, "setFSem", CourseXptn::rangeError, aFSem );
   cFSem = aFSem;
} // Course::setFSem
```

ρίχνει εξαίρεση για την οποίαν θα πρέπει να κάνουμε ορισμένες αλλαγές στην *CourseXptn*:

- Δηλώνουμε άλλο ένα μέλος:

```
   int  errIntVal;
```

- και ορίζουμε άλλον έναν δημιουργό:

```
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
   {  objKey = obk;
      strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      errorCode = ec;
      errIntVal = iv;  }
```

Η *setCompuls()* είναι πολύ απλή:

```
   void setCompuls( bool aCompuls ) { cCompuls = aCompuls; };
```

Να πούμε όμως ότι –για διαχείριση μελών τύπου **bool**– μπορεί να δεις αλλού ένα ζευγάρι μεθόδων:

```
   void setCompuls() {  cCompuls = true;  };
   void clearCompuls() {  cCompuls = false;  };
```

Η αναλλοίωτη μας λέει πώς να γράψουμε τις *setSector()* και *setCateg()*:

```
void Course::setSector( char aSector )
{
   if ( aSector != 'Y' && aSector != 'Π' &&
        aSector != 'M' && aSector != 'Γ' && aSector != 'Ξ' )
      throw CourseXptn( cCode, "setSector",
                        CourseXptn::noSuchSector, aSector );
   cSector = aSector;
} // Course::setSector

void Course::setCateg( string aCateg )
{
   if ( aCateg != "ΜΓΥ" && aCateg != "ΜΕΥ" &&
        aCateg != "ΜΕ" && aCateg != "ΔΟΝ" )
      throw CourseXptn( cCode, "setCateg",
                        CourseXptn::noSuchCateg, aCateg.c_str() );
   strcpy( cCateg, aCateg.c_str() );
} // Course::setSector
```

Οι *setWH()* και *setUnits()* μοιάζουν με την *setFSem()*:

```
void Course::setWH( int aWH )
{
   if ( aWH <= 0 )
      throw CourseXptn( cCode, "setWH", CourseXptn::rangeError, aWH );
   cWH = aWH;
} // Course::setWH

void Course::setUnits( int aUnits )
{
   if ( aUnits <= 0 )
      throw CourseXptn( cCode, "setUnits", CourseXptn::rangeError, aUnits );
   cUnits = aUnits;
} // Course::setUnits
```

και θα έπρεπε να μοιάζουν περισσότερο: θα έπρεπε να υπάρχουν και περιορισμοί εκ των άνω.

Δεν θα γράψουμε *setNoOfStudents()*· ο καλύτερος τρόπος διαχείρισης του μέλους *cNoOfStudents* γίνεται με τις μεθόδους που γράψαμε αρχικώς:

```
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
```

Τώρα όμως θα πρέπει να προβλέψουμε και περίπτωση διαγραφής φοιτητή από κάποιο μάθημα. Για μια τέτοια περίπτωση θα πρέπει να γράψουμε μια:

```
void Course::delete1Student()
{
   if ( cNoOfStudents <= 0 )
      throw CourseXptn( cCode, "delete1Student", CourseXptn::noStudent );
   --cNoOfStudents;
} // Course::delete1Student
```

Η *setPrereq()* είναι κατ' αρχήν –δηλαδή σύμφωνα με όσα λέει η αναλλοίωτη– απλή:

```
void Course::setPrereq( const string& prCode )
{
   if ( prCode.length() != cCodeSz-1 && prCode.length() != 0 )
      throw CourseXptn( cCode, "setPrereq", CourseXptn::keyLen,
                        prCode.c_str() );
   if ( cCode != CourseKey("") && CourseKey(prCode) == cCode )
      throw CourseXptn( cCode, "setPrereq", CourseXptn::autoRef,
                        prCode.c_str() );
   cPrereq = CourseKey( prCode );
} // Course::setPrereq
```

Χρειάζεται όμως ιδιαίτερη μεταχείριση από το πρόγραμμα που τη χρησιμοποιεί διότι, όπως είπαμε, για να βάλουμε εκεί κάποιον κωδικό μαθήματος «*θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων.*» Με αυτό θα ασχοληθούμε στη συνέχεια.

Τέλος –και εκτός «συνταγής» πια– ας έλθουμε στις *save()* και *load()*. Η *save()* παραμένει σχεδόν όπως ήταν. Οι μοναδικές αλλαγές είναι στις γραμμές που φυλάγουν τα μέλη *cCode* και *cPrereq* που θα γίνουν:

```
// . . .
   bout.write( cCode.s, sizeof(cCode.s) );    // κωδικός μαθήματος
// . . .
   bout.write( cPrereq.s, sizeof(cPrereq.s) );   // προαπαιτούμενο
// . . .
```

Στη *load()* θα κάνουμε μεγαλύτερες αλλαγές: θα εφαρμόσουμε αυτά που είδαμε στην §21.6 για να της δώσουμε *ασφάλεια προς τις εξαιρέσεις* επιπέδου *ισχυρής εγγύησης*. Δηλαδή, αφού «*αποπειράται να αλλάξει την τιμή ενός αντικειμένου ... αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.*»

```
void Course::load( istream& bin )
{
   Course tmp;
   bin.read( tmp.cCode.s, cCodeSz );                      // κωδικός μαθήματος
   if ( !bin.eof() )
   {
      bin.read( tmp.cTitle, cTitleSz );                    // τίτλος μαθήματος
      bin.read( reinterpret_cast<char*>(&tmp.cFSem), sizeof(cFSem) );
                                                          // τυπικό εξάμηνο
      bin.read( reinterpret_cast<char*>(&tmp.cCompuls), sizeof(cCompuls) );
                                                       // υποχρεωτικό ή επιλογής
      bin.read( &tmp.cSector, sizeof(cSector) );                  // τομέας
      bin.read( tmp.cCateg, cCategSz );                        // κατηγορία
      bin.read( reinterpret_cast<char*>(&tmp.cWH), sizeof(cWH) );
                                                          // ώρες ανά εβδομάδα
      bin.read( reinterpret_cast<char*>(&tmp.cUnits), sizeof(cUnits));
                                                         // διδακτικές μονάδες
      bin.read( tmp.cPrereq.s, cCodeSz );                 // προαπαιτούμενο
```

```
      bin.read( reinterpret_cast<char*>(&tmp.cNoOfStudents),
                sizeof(cNoOfStudents) );             // αριθ. φοιτ. στο μάθημα
      if ( bin.fail() )
         throw CourseXptn( cCode, "load", CourseXptn::cannotRead );
      *this = tmp;
   } // if ( !bin.eof() ). . .
} // Course::load
```

Τι κάνουμε εδώ; Δηλώνουμε μια βοηθητική μεταβλητή *tmp* –τύπου *Course*– και αποθηκεύουμε σε αυτήν όσα διαβάζουμε. Αν όλα πάνε καλά αντιγράφουμε στο αντικείμενό μας (`*this`) αυτά που είναι αποθηκευμένα στο *tmp*. Δεν θα χρειαστεί να γράψουμε μια *swap*; Δεν είναι υποχρεωτικό! Εδώ δεν έχουμε δεσμεύσει πόρους του συστήματος, ο τελεστής εκχώρησης δουλεύει σωστά. Έτσι η "`*this = tmp`" κάνει τη δουλειά μας και μάλιστα κάπως ταχύτερα από μια "`swap(tmp)`" (αν τη γράψουμε.)

Η νέα εκδοχή της κλάσης είναι:

```
class Course
{ // version 2
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strncpy( s, aKey.c_str(), cCodeSz-1 );  s[cCodeSz-1] = '\0';  }
   }; // CourseKey
   explicit Course( string aCode="", string aTitle="" );
   ~Course() { };
// getters
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   unsigned int getNoOfStudents() const {  return cNoOfStudents; }
   const char* getPrereq() const {  return cPrereq.s;  }
// setters
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) {  cCompuls = aCompuls;  };
   void setSector( char aSector );
   void setCateg( string aCateg );
   void setWH( int aWH );
   void setUnits( int aUnits );
   void clearStudents() {  cNoOfStudents = 0;  }
   void add1Student() {  ++cNoOfStudents;  }
   void delete1Student();
   void setPrereq( const string& prCode );
// other
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
```

```
}; // Course

bool operator!=( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) != 0 );  }

bool operator==( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( !(lhs != rhs) );  }
```

κσι της κλάσης εξαιρέσεων:

```
struct CourseXptn
{ // version 2
   enum { keyLen, rangeError, noSuchSector, noSuchCateg, autoRef, noCourse,
          noStudent, fileNotOpen, cannotWrite, cannotRead };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
      : objKey( obk ),  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // CourseXptn
```

Prj04.3 ... Και ο «Πίνακας Μαθημάτων»

Στο πρόγραμμα του Project 3 παραστήσαμε τον πίνακα μαθημάτων με έναν απλό δυναμικό πίνακα με στοιχεία τύπου *Course*. Η διαχείρισή του έγινε από το πρόγραμμα εφαρμογής και τις συναρτήσεις του.

Εδώ θα τον διαχειριστούμε πιο «νοικοκυρεμένα»: ως αντικείμενο κλάσης

```
class CourseCollection
{
public:
// . . .
private:
   enum { ccIncr = 30 };
   Course* ccArr;
   size_t  ccNOfCourses;
   size_t  ccReserved;
// . . .
}; // CourseCollection
```

με την οποία θα έχουμε την ευκαιρία να δούμε και μερικά νέα(;) πράγματα.

Η *CourseCollection* είναι μια κλάση για τη διαχείριση *συνόλου* που υλοποιούμε με έναν *δυναμικό πίνακα*.

Σύμφωνα με αυτά που είπαμε, θα χρειαστεί να γράψουμε δημιουργό αντιγραφής και τελεστή εκχώρησης. Εδώ όμως προσοχή: Σε οποιοδήποτε πρόγραμμα (μπορούμε να φανταστούμε) που χρησιμοποιεί αντικείμενα αυτής της κλάσης, καθένα από αυτά θα πρέπει να είναι μοναδικό. Διότι αλλιώς έχουμε το εξής πρόβλημα:

- Πόσα αντίγραφα του προγράμματος σπουδών μπορούμε να έχουμε;
- Με ποιο από αυτά θα κάνουμε τι;

Μια λύση του προβλήματος είναι η *απαγόρευση δημιουργίας αντιγράφων κλάσης CourseCollection*. Πρόσεξε πώς:

```
class CourseCollection
{
public:
```

```
// . . .
private:
   enum { ccIncr = 30 };
   Course* ccArr;
   size_t  ccNOfCourses;
   size_t  ccReserved;

   CourseCollection( const CourseCollection& rhs ) { };
   CourseCollection& operator=( const CourseCollection& rhs ) { };
// . . .
}; // CourseCollection
```

Αν στο πρόγραμμά σου (στη `main`) δώσεις:

```
   CourseCollection allCourses;
// . . .
      CourseCollection otherCollection( allCourses );
```

ο μεταγλωττιστής θα βγάλει σφάλμα:

```
'CourseCollection::CourseCollection(const CourseCollection &)' is not accessi-
ble in function main()
```

ή κάτι παρόμοιο, ενώ αν δώσεις:

```
      CourseCollection otherCollection;
// . . .
      otherCollection = allCourses;
```

θα πάρεις:

```
'CourseCollection::operator =(const CourseCollection &)' is not accessible in
function main()
```

ή κάτι παρόμοιο.

Ας έλθουμε τώρα στην *αναλλοίωτη* που περιέχει –κατ' αρχάς– τη συνθήκη διαχείρισης του δυναμικού πίνακα:

$$I_I \equiv (0 \le ccNOfCourses < ccReserved)\ \&\&\ (ccReserved\ \%\ ccIncr == 0)$$

Αλλά έχουμε και επιπλέον περιορισμούς για τα στοιχεία του πίνακα:

- Με τον πίνακα υλοποιούμε ένα *σύνολο*, ας το πούμε *Collection*. Επομένως, δεν θα πρέπει να υπάρχουν δύο ίδια στοιχεία:

$$\forall x, y\text{: } Course \bullet (x, y \in Collection \Rightarrow x \ne y)$$

- Είναι δεκτό ένα στοιχείο του πίνακα με κωδικό μαθήματος μηδενικού μήκους; Όχι! Θα πρέπει λοιπόν να έχουμε:[4]

$$\forall x\text{: } Course \bullet (x \in Collection \Rightarrow strlen(x.cCode) == cCodeSz\text{-}1)$$

- Στην §Prj04.2 λέγαμε για το προαπαιτούμενο: «*θα πρέπει επιπλέον να υπάρχει μάθημα με τέτοιον κωδικό στον πίνακα μαθημάτων. Για να μπορέσεις να βάλεις αυτόν τον περιορισμό στην αναλλοίωτη και τον αντίστοιχο έλεγχο στις μεθόδους θα πρέπει να σχεδιάσεις τις κλάσεις σου έτσι ώστε τα αντικείμενα τύπου Course να υπάρχουν μόνον μέσα σε κάποιον "πίνακα μαθημάτων"*». Εδώ λοιπόν θα πρέπει να βάλουμε:[5]

$$\forall x\text{: } Course \bullet (x \in Collection\ \&\&\ x.cPrereq \ne \text{""})$$
$$\Rightarrow \exists y\text{: } Course \bullet (y \in Collection\ \&\&\ y.cCode == x.cPrereq)$$

Η αναλλοίωτη είναι:

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

όπου το πρώτο κομμάτι είναι:

$$I_D \equiv (\forall x, y\text{: } Course \bullet (x, y \in Collection \Rightarrow x \ne y))\ \&\&$$
$$(\forall x\text{: } Course \bullet (x \in Collection \Rightarrow strlen(x.cCode) == cCodeSz\text{-}1))\ \&\&$$

---

[4] Στις Βάσεις Δεδομένων αυτός ο περιορισμός ονομάζεται **περιορισμός** (constraint) **ακεραιότητας οντότητας** (entity integrity)...

[5] ... και αυτός περιορισμός **ακεραιότητας αναφοράς** (referential integrity).

$$(\forall x\text{: } Course \bullet (x \in Collection \ \&\& \ x.cPrereq \neq \texttt{""})$$
$$\Rightarrow \exists y\text{: } Course \bullet (y \in Collection \ \&\& \ y.cCode \texttt{ == } x.cPrereq))$$

Παρ' όλο που προσπαθήσαμε να γράψουμε την αναλλοίωτη κάπως αφηρημένα είναι σαφές ότι παίρνουμε υπόψη μας ότι το *cCode* είναι κλειδί.

Αν πάρουμε υπόψη μας ότι το σύνολο υλοποιείται με τα πρώτα *ccNOfCourses* στοιχεία του πίνακα ccArr η I_D γίνεται:

$$I_D \equiv (\forall j, k\text{: } \texttt{[}0 \texttt{ .. } ccNOfCourses\texttt{)} \bullet (j \neq k \Rightarrow \text{ccArr}\texttt{[}j\texttt{].}cCode \neq \text{ccArr}\texttt{[}k\texttt{].}cCode)) \ \&\&$$
$$(\forall k\text{: } \texttt{[}0 \texttt{ .. } ccNOfCourses\texttt{)} \bullet strlen(\text{ccArr}\texttt{[}k\texttt{].}cCode) \texttt{ == } cCodeSz\text{-}1) \ \&\&$$
$$(\forall k\text{: } \texttt{[}0 \texttt{ .. } ccNOfCourses\texttt{)} \bullet (\text{ccArr}\texttt{[}k\texttt{].}cPrereq \neq \texttt{""} \Rightarrow$$
$$\exists j\text{: } \texttt{[}0 \texttt{ .. } ccNOfCourses\texttt{)} \bullet \text{ccArr}\texttt{[}j\texttt{].}cCode \texttt{ == } \text{ccArr}\texttt{[}k\texttt{].}cPrereq))$$

Κατά τα *άλλα*:

- Έχουμε ερήμην δημιουργό:

```
CourseCollection::CourseCollection()
{
   try
   {
      ccReserved = ccIncr;
      ccArr = new Course[ ccReserved ];
      ccNOfCourses = 0;
   }
   catch( bad_alloc& )
   {
      throw CourseCollectionXptn( "CourseCollection",
                                  CourseCollectionXptn::allocFailed);
   }
} // CourseCollection::CourseCollection
```

- Πρέπει να γράψουμε *καταστροφέα*:

```
   ~CourseCollection()  {  delete[] ccArr;  };
```

- Θα γράψουμε συναρτήσεις "*get*"…

```
   size_t getNOfCourses() const {  return ccNOfCourses;  }
   const Course* getArr() const {  return ccArr;  }
```

- … αλλά δεν θα γράψουμε συναρτήσεις "*set*".

Θα πρέπει να γράψουμε και μεθόδους διαχείρισης των στοιχείων του πίνακα που θα μοιάζουν με αυτές που γράψαμε για τη *Route* (§20.7.2.3). Ξαναδές τι κάναμε με τον πίνακα rAllStops:

- Ο ccArr έχει μια σημαντική ομοιότητα με τον rAllStops: *και οι δύο χρησιμοποιούνται για την παράσταση συνόλων*. Έτσι, όπως είχαμε τις *find1RouteStop()*, *get1RouteStop()*, *deleteRouteStop()* – *erase1RouteStop()* και *addRouteStop()* – *insert1RouteStop()* θα πρέπει να γράψουμε τις *find1Course()*, *get1Course()*, *delete1Course()* – *erase1Course()* και *add1Course()* – *insert1Course()* αντιστοίχως.
- Τα στοιχεία του συνόλου που παριστάνει ο rAllStops έχουν, ανά δύο, διαφορετικό *sName* και διαφορετική *sDist*. Στην περίπτωση του ccArr τα στοιχεία έχουν, ανά δύο, διαφορετικό *cCode*. Ο κωδικός μαθήματος είναι άλλο ένα χαρακτηριστικό παράδειγμα *υποκατάστατου κλειδιού* (§15.5.1), όπως είναι ο αριθμός μητρώου για τη *Student*.
- Στην περίπτωση του rAllStops προτιμήσαμε να επιφορτώσουμε τον "`!=`" με σύγκριση στο μέλος *sDist* διότι είχαμε να χειριστούμε την ταξινόμηση του πίνακα σύμφωνα με την απόσταση από την αφετηρία. Αυτό μας στέρησε τη δυνατότητα χρήσης της *linSearch* –από τη `MyTmpltLib`– για αναζητήσεις στάσεων με βάση το όνομά τους. Εδώ έχουμε να κάνουμε αναζητήσεις με βάση τον κωδικό μαθήματος. Αν θέλουμε να χρησιμοποιήσουμε τη *linSearch()* μπορούμε να επιφορτώσουμε τον "`!=`" για την *Course* με σύγκριση των κωδικών μόνο. Η επιφόρτωση θα μπορούσε να γίνει, όπως στη *Date* (και τη *Student*), με μια καθολική συνάρτηση:

```
bool operator!=( const Course& a, const Course& b )
```

```
{  return ( strcmp(a.getCode(), b.getCode()) != 0 );  }
```

Αυτή περνάει από τον μεταγλωττιστή και το πρόγραμμά μας δουλεύει μια χαρά. Ας πούμε τώρα ότι –για κάποιον λόγο– θέλουμε να αλλάξουμε τη λογική σύγκρισης κλειδιών και αντικειμένων. Θα πρέπει να αλλάξουμε τις συναρτήσεις επιφόρτωσης του "`!=`" της *CourseKey* και της *Course* που υλοποιούν την ίδια λογική. Φυσικά, αυτό εισάγει στο λογισμικό μας μια δυνατότητα για μελλοντικό σφάλμα. Αυτό αποφεύγεται αν ορίσουμε:

```
bool operator!=( const Course& lhs, const Course& rhs )
{  return ( Course::CourseKey(lhs.getCode()) !=
            Course::CourseKey(rhs.getCode()) );  }
```

οπότε η λογική της σύγκρισης υλοποιείται μια φορά μόνον, για τον τύπο *CourseKey*. Πέρα από αυτό, εδώ λέμε ξεκάθαρα ότι δύο αντικείμενα τύπου *Course* είναι διαφορετικά αν και μόνον αν έχουν διαφορετικούς κωδικούς.

- Η –μικρότερης σημασίας– διαφορά του ccArr από τον rAllStops είναι η ταξινόμηση των στοιχείων του rAllStops. Αφού ο ccArr δεν είναι ταξινομημένος δεν χρειάζεται να κάνουμε πολλές μετακινήσεις στοιχείων στη διαγραφή και την εισαγωγή. Στην §20.7.2 δώσαμε ένα σχέδιο για τη *διαγραφή* σε μια τέτοια περίπτωση.

  Αρχίζουμε από την *find1Course()* που θα γίνει:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
```

όπου τώρα:

```
int CourseCollection::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( ccArr, ccNOfCourses, 0, ccNOfCourses-1, Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Πρόσεξε ότι:

- Και εδώ η *findNdx()* επιστρέφει τον δείκτη του στοιχείου (ή "-1" αν η αναζήτηση αποτύχει) ενώ η *find1Course()* είναι υλοποίηση της σχέσης "∈" και επιστρέφει `true` ή `false`.
- Και εδώ θα δηλώσουμε τη *findNdx()* στην περιοχή `private` με ίδιο σκεπτικό που είδαμε στην περίπτωση της *Route* (§20.7.2.3).
- Αποκλείεται να κληθεί ο δημιουργός της *Course* αν η τιμή της *code* έχει «παράνομο» μήκος.

  Ας δούμε τώρα την *erase1Course()*. Μετατρέπουμε το σχέδιο που δώσαμε στην §20.7.2:

```
void CourseCollection::delete1Course( string code )
{
   if ( υπάρχει μάθημα με κωδικό code στη θέση ndx )
   {
      αντίγραψε στη θέση ndx το τελευταίο στοιχείο του πίνακα
      --ccNOfCourses;
   }
}
```

και γράφουμε τη μέθοδο:

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // code found
   {
      ccArr[ndx] = ccArr[ccNOfCourses-1];
      --ccNOfCourses;
   }
```

```
} // CourseCollection::delete1Course
```

Όπως θα δούμε στη συνέχεια, αυτή η μέθοδος μπορεί να οδηγήσει σε παραβίαση της αναλλοίωτης. Στη συνέχεια θα τη συμπληρώσουμε και θα τη διασπάσουμε σε δύο μεθόδους.

Η *insert1Course()* μικρή σχέση έχει με την *insert1RouteStop()*· είναι πολύ απλούστερη. Αν δεν υπάρχει το στοιχείο που θέλουμε να εισαγάγουμε το αντιγράφουμε στο τέλος του πίνακα. Φυσικά, «μεγαλώνουμε» τον πίνακα αν δεν έχει χώρο για το νέο στοιχείο:

```
void CourseCollection::insert1Course( const Course& aCourse )
{
   if ( ccReserved <= ccNOfCourses+1 )
   {
      try {  renew( ccArr, ccNOfCourses, ccReserved+ccIncr );
             ccReserved += ccIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw CourseCollectionXptn( "insert1Course",
                                     CourseCollectionXptn::allocFailed );
      }
   }
   ccArr[ccNOfCourses] = aCourse;
   ++ccNOfCourses;
} // CourseCollection::insert1Course
```

Prj04.3.1 Οι Μέθοδοι *add1Course()* και *delete1Course()*

Η δουλειά που κάνει η *insert1Course()* δεν είναι αρκετή για την περίπτωσή μας αφού δεν συμμορφώνεται με την αναλλοίωτη. Έτσι, θα πρέπει να γράψουμε μια άλλη μέθοδο, ας την πούμε *add1Course()*, που θα καλεί την *insert1Course()* αφού κάνει τους εξής ελέγχους:

- Ο κωδικός μαθήματος θα πρέπει να έχει μήκος `cCodeSz-1`:

```
   if ( strlen(aCourse.getCode()) != cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::entity);
```

- Ο πίνακας δεν θα πρέπει να έχει στοιχείο με κωδικό ίδιο με αυτόν του εισαγόμενου:

```
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx >= 0 )
      throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::key,
                                  aCourse.getCode() );
```

  Αυτό όμως είναι πολύ «αυστηρό»! Αν υπάρχει το στοιχείο δεν το εισάγουμε και τελειώσαμε· η ρίψη εξαίρεσης είναι υπερβολή.

- Αν το εισαγόμενο στοιχείο έχει προαπαιτούμενο αυτό θα πρέπει να υπάρχει ήδη στον πίνακα:

```
   if ( strcmp(aCourse.getPrereq(), "") != 0 )       // υπάρχει προαπαιτούμενο
   {
      int ndx( findNdx(aCourse.getPrereq()) );
      if ( ndx < 0 )              // δεν βρέθηκε το κλειδί του προαπαιτούμενου
         throw CourseCollectionXptn( "add1Course", CourseCollectionXptn::ref,
                                     aCourse.getPrereq() );
   }
```

- Το εισαγόμενο στοιχείο θα πρέπει να μην έχει εγγεγραμμένους φοιτητές. Δηλαδή θα πρέπει να ελέγχουμε το *cNoOfStudents*; Όχι, θα το μηδενίζουμε! Βέβαια, δεν θα πειράζουμε την παράμετρο (περνάει ως "`const Course& aCourse`") αλλά αυτό που εισάχθηκε στον πίνακα:

```
      ndx = findNdx( aCourse.getCode() );
      ccArr[ndx].clearStudents();
```

  Η *add1Course()* θα είναι:

```
void CourseCollection::add1Course( const Course& aCourse )
{
```

```
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )// υπάρχει
      {                                          // προαπαιτούμενο
         int ndx( findNdx(aCourse.getPrereq()) );
         if ( ndx < 0 )   // δεν βρέθηκε το κλειδί προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::ref,
                                        aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      insert1Course( aCourse );
      ndx = findNdx( aCourse.getCode() );
      ccArr[ndx].clearStudents();
   }
} // CourseCollection::add1Course
```

Όπως καταλαβαίνεις για να μην έχουμε δυσάρεστες εκπλήξεις θα πρέπει να «κρύψουμε» (**private**) την *insert1Code()*.

Πρόβλημα συμμόρφωσης με την αναλλοίωτη μπορεί να έχουμε και με τη διαγραφή: Αν διαγράψουμε κάποιο στοιχείο που είναι προαπαιτούμενο άλλου (-ων) στοιχείου (-ων). Θα πρέπει λοιπόν να «κρύψουμε» και την *erase1Course()* και να γράψουμε μια *delete1Course()*, που θα καλεί την *erase1Course()* αφού κάνει τους απαραίτητους ελέγχους.

Και κάτι ακόμη: όπως θα θυμάσαι από το Project 3, υπάρχει και ο *Πίνακας Δηλώσεων Μαθημάτων*. Αν το μάθημα που θέλουμε να σβήσουμε έχει *cNoOfStudents* > 0 θα πει ότι

- υπάρχουν στοιχεία του *Πίνακα Δηλώσεων* με τον κωδικό του,
- υπάρχουν αντικείμενα κλάσης *Student* με τον κωδικό του στον πίνακα μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.

Επομένως δεν μπορούμε να σβήσουμε το μάθημα!

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // υπάρχει
   {
      ccArr[ccNOfCourses] = Course();
      ccArr[ccNOfCourses].setPrereq( code ); // φρουρός
      int k(0);
      while ( strcmp(ccArr[k].getPrereq(), code.c_str()) != 0 )
         ++k;
      if ( k < ccNOfCourses )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::cannotDel,
                                     code.c_str() );
      int enrStdnt( ccArr[ndx].getNoOfStudents() );
      if ( enrStdnt > 0 ) // υπάρχουν εγγραφές φοιτητών
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::enrollRef,
                                     code.c_str(), enrStdnt );
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Αν βρούμε ότι στον πίνακα υπάρχει μάθημα με τον συγκεκριμένο κωδικό (*code*) κάνουμε μια γραμμική αναζήτηση με φρουρό στο μέλος *cPrereq*. Αν βρούμε έστω και ένα στοιχείο του πίνακα που έχει ως προαπαιτούμενο μάθημα με κωδικό *code* ρίχνουμε εξαίρεση.

Ακόμη ρίχνουμε εξαίρεση αν βρούμε `ccArr[ndx].getNoOfStudents() > 0`.

Σημείωση:▶
Αργότερα θα δούμε πώς μπορείς να χειρισθείς με άλλον τρόπο τη διαγραφή μαθήματος στο οποίο υπάρχουν εγγραφές φοιτητών.◀

Αφού οι έλεγχοι πέρασαν στη *delete1Course()*, η *erase1Course()* απλουστεύεται:

```
void CourseCollection::erase1Course( int ndx )
{
   ccArr[ndx] = ccArr[ccNOfCourses-1];
   --ccNOfCourses;
} // CourseCollection::erase1Course
```

Πρόσεξε ότι τώρα, αφού την καλούμε όταν ξέρουμε ότι το μάθημα υπάρχει σίγουρα, την τροφοδοτούμε με τον δείκτη του προς διαγραφή στοιχείου.

Prj04.3.2 Οι *save()* και *load()*

Οι *save()* και *load()* χρησιμοποιούν τις αντίστοιχες της *Course*. Πρώτα η

```
void CourseCollection::save( ofstream& bout )
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save", CourseCollectionXptn::fileNotOpen);
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( int k(0); k < ccNOfCourses; ++k )
      ccArr[k].save( bout );
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save", CourseCollectionXptn::cannotWrite);
} // CourseCollection::save
```

Η *load()* θέλει πιο πολλή προσοχή, αφού θα πρέπει να έχει *ασφάλεια προς τις εξαιρέσεις* επιπέδου *ισχυρής εγγύησης*. Δηλαδή, όπως λέγαμε στην §21.6, αφού «*αποπειράται να αλλάξει την τιμή ενός αντικειμένου ... αν αποτύχει θα πρέπει να αφήνει την παλιά τιμή χωρίς αλλαγές.*»

Για να το διασφαλίσουμε θα χρησιμοποιήσουμε ένα τοπικό αντικείμενο

```
   CourseCollection tmp;
```

όπου θα αποθηκεύουμε τις τιμές που διαβάζουμε από το αρχείο. Αν όλα πάνε καλά, θα ανταλλάξουμε την τιμή του *tmp* με την τιμή του `*this`, με χρήση της

```
void CourseCollection::swap( CourseCollection& rhs )
{
   Course* sv( ccArr );
   ccArr = rhs.ccArr;  rhs.ccArr = sv;
   std::swap( ccNOfCourses, rhs.ccNOfCourses );
   std::swap( ccReserved, rhs.ccReserved );
} // CourseCollection::swap
```

Το πρώτο που κάνουμε είναι να διαβάσουμε το πλήθος των στοιχείων του πίνακα σε μια τοπική μεταβλητή:

```
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(ccNOfCourses) );
```

Η τιμή της *n* μας δίνει το πλήθος των τιμών τύπου *Course* που ακολουθούν στο αρχείο. Τις διαβάζουμε ως εξής:

```
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
```

Πρόσεξε ότι εδώ καλούμε την *insert1Course()* και όχι την *add1Course()* θεωρώντας –ως συνήθως– ότι το περιεχόμενο του μη μορφοποιημένου αρχείου δεν έχει σφάλματα.

Αν η εκτέλεση της **for** τελειώσει επειδή κάτι δεν πήγε καλά ρίχνουμε εξαίρεση. Αλλιώς δίνουμε στο αντικείμενό μας την τιμή του *tmp* με την (ασφαλή) *swap*:

```
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
```

Με την ολοκλήρωση εκτέλεσης της *load()* καταστρέφεται το *tmp* και ανακυκλώνεται η δυναμική μνήμη που έχει δεσμεύσει. Το ίδιο θα συμβεί και στην περίπτωση που η εκτέλεση της *load()* διακόπτεται επειδή ρίχνεται κάποια εξαίρεση. Εδώ έχουμε εφαρμογή της τεχνικής *RAII*.

Ολόκληρη η *load*:

```
void CourseCollection::load( ifstream& bin )
{
   CourseCollection tmp;
   unsigned int n;

   bin.read( reinterpret_cast<char*>(&n), sizeof(ccNOfCourses) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Prj04.3.3 Απώλειες Πρόσβασης

Κρύβοντας σε περιοχή **private** όλα τα αντικείμενα *Course* αχρηστεύουμε τις ανοιχτές μεθόδους τους. *Αυτά τα αντικείμενα θα παραμείνουν όπως είναι όταν τα βάζουμε για πρώτη φορά*! Δεν μπορούμε να χρησιμοποιήσουμε τη διαδικασία «*get-delete-edit-add*» που είδαμε στη *Route*; Όχι! Θα πάρουμε εξαίρεση

- αν προσπαθήσουμε να σβήσουμε ένα μάθημα που είναι προαπαιτούμενο για κάποιο άλλο·
- αν προσπαθήσουμε να σβήσουμε ένα μάθημα στο οποίο έχουν εγγραφεί φοιτητές (αυτό θα το φροντίσουμε στη συνέχεια).

Ναι, αλλά –πέρα από ανάγκες διόρθωσης λανθασμένων στοιχείων– θα χρειαστεί να αυξάνουμε (μειώνουμε) τον αριθμό των φοιτητών που γράφονται σε (διαγράφονται από) ένα μάθημα. Τι κάνουμε σε αυτήν την περίπτωση; Γράφουμε για την *CourseCollection* ενδιάμεσες μεθόδους που επιτρέπουν την χρήση των μεθόδων των αντικειμένων *Course*. Στην περίπτωσή μας θα δηλώσουμε τις:

```
   void add1Student( string code );
   void delete1Student( string code );
```

και θα τις ορίσουμε:

```
void CourseCollection::add1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
```

```
   ccArr[ndx].add1Student();
} // CourseCollection::add1Student

void CourseCollection::delete1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ccArr[ndx].delete1Student();
} // CourseCollection::delete1Student
```

Τα ονόματα είναι αυτά των αντίστοιχων μεθόδων της *Course*·αν δεν σου αρέσουν άλλαξέ τα.

Παρόμοιες ενδιάμεσες μεθόδους (της *CourseCollection*) μπορείς να γράψεις και για άλλες μεθόδους της *Course*· *μη γράψεις για τη setCode*. Να θυμάσαι ότι γενικώς, από τη στιγμή που το αντικείμενο μπήκε στη συλλογή, *το κλειδί δεν αλλάζει*. Αν αλλάξεις κάποιον κωδικό μαθήματος θα μείνουν «ξεκρέμαστα»

- τα αντικείμενα του πίνακα που έχουν το μάθημα ως προαπαιτούμενο και
- τα αντικείμενα του *Πίνακα Δήλωσης Μαθημάτων* για εγγραφή σε αυτό το μάθημα.

Prj04.3.4 Επιφορτώνουμε τον "[]";

Πιθανότατα θα αναρωτηθείς: «Αφού ένα αντικείμενο της *CourseCollection* είναι ένας πίνακας στοιχείων τύπου *Course* γιατί να μην επιφορτώσουμε τον τελεστή "`[]`";»

Ναι, ένα αντικείμενο της *CourseCollection* είναι ένας πίνακας στοιχείων τύπου *Course* αλλά «κρύβουμε» αυτό το γεγονός για να μην ενθαρρύνουμε την ανάπτυξη εφαρμογών που να στηρίζονται στην παράσταση των δεδομένων του συνόλου. Έτσι, η *get1Course()* υλοποιείται ως εξής:

```
const Course& CourseCollection::get1Course( string code ) const
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return ccArr[ndx];
} // CourseCollection::get1Course
```

Ο δείκτης *ndx* είναι τοπική μεταβλητή και το πρόγραμμα που χρησιμοποιεί αυτήν τη μέθοδο παίρνει αυτό που ζητάει δίνοντας μια τιμή τύπου *string*.

Αν θέλεις μπορείς να επιφορτώσεις τον "`[]`" ως εναλλακτικό τρόπο γραφής της *get1Course*:

```
const Course& CourseCollection::operator[]( string code ) const
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return ccArr[ndx];
} // CourseCollection::operator[]
```

Αλλά ποιο από τα παρακάτω είναι προτιμότερο:

```
   get1C = allCourses.get1Course( "EY02010" );
   get1C = allCourses["EY02010"];
```

Το δεύτερο φαίνεται λίγο «παράξενο»: μεταξύ των "`[`" και "`]`" έχουμε συνηθίσει να βλέπουμε παράσταση που δίνει έναν φυσικό αριθμό, τον δείκτη για το στοιχείο του πίνακα.

Υπάρχει και ένα άλλο πρόβλημα: Συνήθως ο τύπος του αποτελέσματος της συνάρτησης επιφόρτωσης του "`[]`" είναι *τύπος αναφοράς χωρίς το* "`const`". Έτσι μπορούμε να αλλάζουμε την τιμή του στοιχείου που μας δίνει. Εδώ δεν θέλουμε να δώσουμε τέτοια δυνατότητα.

Όπως βλέπεις, μια τέτοια επιφορτωση θα ήταν ανορθόδοξη από πολλές απόψεις και θα προτιμήσουμε να την αποφύγουμε. Αργότερα θα δούμε ότι κάτι τέτοιο γίνεται στην STL.

Prj04.4 Περί Διαγραφών

Είδαμε πιο πάνω την *CourseCollection::delete1Course()* και μην αμφιβάλλεις για τη συνέχια: θα δούμε και *StudentCollection::delete1Student()* και *StudentInCourseCollection::delete1StudentInCourse()*. Κάνουμε βεβαίως ελέγχους, αλλά σβήνουμε μάλλον εύκολα.

Σκέψου όμως τα εξής: τα διαγραφόμενα μπορεί να περιέχουν στοιχεία που έχουν εισαχθεί με το χέρι· αυτή είναι μια «ακριβή» διαδικασία. Τέτοια στοιχεία δεν τα σβήνουμε «τελεσιδίκως» διότι είναι πιθανό να χρειαστεί να εισαχθούν εκ νέου στους πίνακές μας (όπως είναι ή μετά από κάποια διόρθωση). Να θυμάσαι ότι γενικώς:

- ***Σπανίως σβήνουμε δεδομένα που έχουν εισαχθεί σε μια ΒΔ με το χέρι ή άλλη χρονοβόρα –και επομένως «ακριβή»– διαδικασία.***

Αυτό που μπορείς να κάνεις είναι να γράψεις τις μεθόδους διαγραφής έτσι ώστε τα αντικείμενα που θα τα αφαιρέσει από κάποιον πίνακα να μεταφέρονται σε έναν άλλον πίνακα με την ίδια (περίπου) δομή. Τι εννοούμε με το «περίπου»; Συνήθως κάθε αντικείμενο που εισάγεται στα «διαγραφέντα» συνοδεύεται από τον χρόνο της διαγραφής.

Prj04.5 Η Κλάση *Student*

Η *Student* θα είναι τώρα:

```
class Student
{
public:
// . . .
private:
   enum { sNameSz = 20 };
   enum { sIncr = 3 };
   unsigned int       sIdNum;                  // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];
   unsigned int       sWH;                     // ώρες ανά εβδομάδα
   size_t             sNoOfCourses;            // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
   size_t             sReserved;
}; // Student
```

Το βέλος *sCourses* θα δείχνει τον δυναμικό πίνακα όπου «*θα αποθηκεύονται ... οι κωδικοί μαθημάτων στα οποία έχει εγγραφεί ο φοιτητής.*»

Η –αντίστοιχη της *Student*– κλάση εξαιρέσεων, όπως –σχεδόν– την έχουμε από το Project 3 είναι:

```
struct StudentXptn
{
   enum { incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, int ev = 0 )
```

```
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Η μόνη διαφορά –προς το παρόν– είναι η εισαγωγή ενός ακόμη μελους, του *objKey* και η αλλαγή στον δημιουργό ώστε να το χειριστεί.

Τώρα, ας εφαρμόσουμε τη συνταγή μας:

- **ΑΝ**: Ας ξεκινήσουμε με την αναλλοίωτη της κλάσης:

$$0 \leq sIdNum \;\&\&\; strlen(sSurname) < sNameSz \;\&\&\; strlen(sFirstname) < sNameSz \;\&\&\; sWH \geq 0 \;\&\&$$
$$(\forall k\colon [0..sNoOfCourses) \bullet (strlen(\text{sCourses}[k]) == Course::cCodeSz-1)) \;\&\&$$
$$(\forall j, k\colon [0..sNoOfCourses) \bullet (j \neq k \Rightarrow \text{sCourses}[j] \neq \text{sCourses}[k])) \;\&\&$$
$$(0 \leq sNoOfCourses < sReserved) \;\&\&\; (sReserved \,\%\, sIncr == 0)$$

Να παρατηρήσουμε τα εξής:

1) Το *sIdNum* μπορεί να πάρει τιμή "`0`" μόνον από τον ερήμην δημιουργό σε δήλωση της μορφής:

```
   Student s1;
```

2) Για το μέλος *sIdNum* χρειαζόμαστε πιο «περιοριστική» συνθήκη· με τη "0 <= *sIdNum*" γίνεται δεκτός ο αριθμός μητρώου "`1000000`" που είναι μάλλον απίθανο να υπάρχει για κάποιο τμήμα.

3) Όπως στην περίπτωση του *cTitle* της *Course* έτσι και για τα *sSurname* και *sFirstname*, δεν θα ρίχνουμε εξαίρεση αν μας έλθει τιμή με μήκος *sNameSz* ή μεγαλύτερο. Απλώς θα αποθηκεύουμε τους πρώτους *sNameSz*-1 χαρακτήρες.

- **ΔΕ**: Θα χρειαστούμε φυσικά έναν ερήμην δημιουργό.

```
Student:: Student( int aIdNum )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
   sWH = 0;
   try  {  sCourses = new Course::CourseKey[ sIncr ];  }
   catch( bad_alloc )
   {  throw StudentXptn( sIdNum, "Student", StudentXptn::allocFailed );  }
   sReserved = sIncr;
   sNoOfCourses = 0;
}; // Student()
```

Πρόσεξε τη διαφορά των δύο εξαιρέσεων:

- Η πρώτη ρίχνεται αν η *aIdNum* έχει παράνομη τιμή οπότε και δεν μπορούμε να βάλουμε τιμή στο *sIdNum* που είναι κλειδί.
- Στη δεύτερη περίπτωση το *sIdNum* έχει τιμή (που πήρε από την *aIdNum*.)

Prj04.5.1 Ο «Κανόνας των Τριών»

- **ΔΑ**: Θα χρειαστούμε και έναν δημιουργό αντιγραφής; Ο πίνακας μαθημάτων θα είναι δυναμικός. Έτσι, τα αντικείμενα της κλάσης μας δεσμεύουν πόρους του συστήματος (μνήμη) και θα χρειαστεί να γράψουμε δημιουργό αντιγραφής αφού αυτός που μας δίνει αυτομάτως ο μεταγλωττιστής δεν είναι σωστός.
- **ΤΕ**, **ΚΑ**: Για τον ίδιο λόγο (δυναμικός πίνακας) θα χρειαστεί να γράψουμε τελεστή εκχώρησης και καταστροφέα.

  Ο καταστροφέας είναι απλός:

```
   ~Student() {  delete[] sCourses;  };
```

Και ο δημιουργός αντιγραφής:

```
Student::Student( const Student& rhs )
```

```
{
   sIdNum = rhs.sIdNum;
   strcpy( sSurname, rhs.sSurname );
   strcpy( sFirstname, rhs.sFirstname );
   sWH = rhs.sWH;
   try  {  sCourses = new Course::CourseKey[ rhs.sReserved ];  }
   catch( bad_alloc )
   {  throw StudentXptn( sIdNum, "Student", StudentXptn::allocFailed );  }
   sReserved = rhs.sReserved;
   for ( int k(0); k < rhs.sNoOfCourses; ++k )
      sCourses[k] = rhs.sCourses[k];
   sNoOfCourses = rhs.sNoOfCourses;
}; // Student( const Student& rhs )
```

Και ο (αντιγραφικός) τελεστής εκχώρησης:

```
Student& Student::operator=( const Student& rhs )
{
   if ( &rhs != this )
   {
      try {  Student tmp( rhs );
             swap( tmp );  }
      catch( StudentXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
}; // Student( const Student& rhs )
```

όπου

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );

   std::swap( sWH, rhs.sWH );
   std::swap( sNoOfCourses, rhs.sNoOfCourses );

   Course::CourseKey* svck( sCourses );
   sCourses = rhs.sCourses;  rhs.sCourses = svck;

   std::swap( sReserved, rhs.sReserved );
} // Student::swap
```

Prj04.5.2 Μέθοδοι "*get*" και "*set*"

GE: Θα χρειαστούμε μεθόδους "*get*" για όλα τα μέλη εκτός από το *sReserved* («μυστικό» της υλοποίησης):

```
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
   unsigned int getWH() const {  return sWH;  }
   unsigned int getNoOfCourses() const { return sNoOfCourses; }
   const Course::CourseKey* getCourses() const { return sCourses;}
```

SE: Θα χρειαστούμε μεθόδους "*set*" για όλα τα μέλη –κατ' αρχήν– εκτός από το *sReserved*.

Πρώτα η

```
void Student::setIdNum( int aIdNum )
{
```

```
   if ( aIdNum <= 0 )
      throw StudentXptn( sIdNum, "setIdNum", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
} // Student::setIdNum
```

Πρόσεξε ότι εδώ δεν επιτρέπουμε τιμή "0", που επιτρέπεται (μόνο) στον ερήμην δημιουργό.

Άλλες δύο, οι *setSurname()* και *setFirstname()*, είναι απλές:

```
void Student::setSurname( string aSurname )
{
   strncpy( sSurname, aSurname.c_str(), sNameSz-1 );
   sSurname[sNameSz-1] = '\0';
} // Student::setSurname

void Student::setFirstname( string aFirstname )
{
   strncpy( sFirstname, aFirstname.c_str(), sNameSz-1 );
   sFirstname[sNameSz-1] = '\0';
} // Student::setSurname
```

Στην περίπτωση (μάλλον απίθανη) που θα έλθει κάποιο επώνυμο ή όνομα με περισσότερους από 19 χαρακτήρες θα αντιγραφούν μόνον οι πρώτοι 19 και φυσικά δεν θα ρίξουμε εξαίρεση. Πάντως, καλό θα είναι να μάθει ο χρήστης του προγράμματος-πελάτη, με κάποιον τρόπο, ότι μια μέθοδος της κλάσης αλλοίωσε τα δεδομένα του!

Θα μπορούσαμε να γράψουμε και *μια* μέθοδο (*setCourses*) που θα δίνει τιμή στο *sNoOfCourses* και τον πίνακα sCourses. Δεν θα το κάνουμε· αν θέλεις κάνε το σαν άσκηση. Πάντως, μια τέτοια μέθοδος δεν θα ήταν και τόσο χρήσιμη αφού στη συνέχεια θα γράψουμε τις *insert1Course()* και *add1Course()*.

Στις μεθόδους "*set*" θα περιλάβουμε και την

```
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
```

που είχαμε στο Project 3. Αυτή παραμένει όπως ήταν! Και ο πίνακας; Ο πίνακας παραμένει όπως είναι, η τιμή του *sReserved* δεν αλλάζει, αλλά η τιμή του *sNoOfCourses* (== 0) μας λέει ότι δεν υπάρχουν στοιχεία για επεξεργασία. Θα μπορούσαμε βεβαίως να επιστρέψουμε τη μνήμη του δυναμικού πίνακα και να πάρουμε νέα με *sIncr* στοιχεία μόνον. Δεν αξίζει τον κόπο...

Prj04.5.3 Μέθοδοι για τα Στοιχεία του Πίνακα

Κατ' αρχήν μπορούμε να χρησιμοποιήσουμε ως οδηγό τις αντίστοιχες μεθόδους της *CourseCollection* –και αυτό θα κάνουμε– εστιάζοντας την προσοχή μας στις διαφορές.

Η βασική μέθοδος θα είναι η *findNdx()* που τροφοδοτείται με έναν κωδικό μαθήματος και μας επιστρέφει τη θέση του στον πίνακα ή, αν δεν υπάρχει, "–1". Την αντιγράφουμε (σχεδόν) από την *CourseCollection*:

```
int Student::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( sCourses, sNoOfCourses, 0, sNoOfCourses-1,
                       Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Η *find1Course()* θα είναι:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
```

Φυσικά, κάθε πρόγραμμα-πελάτης μπορεί να χρησιμοποιεί μόνον τη δεύτερη ενώ η πρώτη θα είναι «κρυμμένη» στην περιοχή `private`.

Θα γράψουμε *get1Course()*; Δεν απαγορεύεται αλλά δεν έχει και νόημα, αφού θα είναι μια συνάρτηση που θα τροφοδοτείται με έναν κωδικό και θα επιστρέφει τον ίδιο κωδικό.

Θα υλοποιήσουμε τώρα την *insert1Course()*:

```
void Student::insert1Course( const Course::CourseKey& aCode )
{
   if ( sReserved <= sNoOfCourses+1 )
   {
      try {  renew( sCourses, sNoOfCourses, sReserved+sIncr );
             ccReserved += ccIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw StudentXptn( sIdNum, "insert1Course",
                                    StudentXptn::allocFailed );
      }
   }
   sCourses[sNoOfCourses] = aCode;
   ++sNoOfCourses;
} // Student::insert1Course
```

Πρόσεξε όμως ότι η μέθοδος αυτή

- Δεν εξασφαλίζει τη μοναδικότητα των κωδικών στον πίνακα.
- Δεν ενημερώνει το μέλος *sWH*.
- Δεν ελέγχει αν υπάρχει στον πίνακα μαθημάτων μάθημα με τέτοιον κωδικό (*ακεραιότητα αναφοράς*).

Για να αποτρέψουμε απρόσεκτη –και εν δυνάμει καταστροφική– χρήση της μεθόδου θα την κρύψουμε, όπως κάναμε και με τη *insert1Course()* της *CourseCollection*· για εξωτερική χρήση θα ξαναγράψουμε την *add1Course()* έτσι που να αντιμετωπίζει τα πρώτα δύο προβλήματα. Αργότερα θα αντιμετωπίσουμε και το τρίτο πρόβλημα.

```
void Student::add1Course( const Course& oneCourse )
{
   if ( findNdx(oneCourse.getCode()) < 0 )
   {
      insert1Course( Course::CourseKey(oneCourse.getCode()) );
      sWH += oneCourse.getWH();
   }
} // Student::add1Course
```

Πρόσεξε ότι, παρ' όλο που στην *insert1Course()* θα περάσουμε μόνον τον κωδικό, τροφοδοτούμε τη μέθοδο με ολόκληρο το αντικείμενο κλάσης *Course*· και αυτό διότι στην επόμενη γραμμή παίρνουμε από αυτό και τις ώρες εβδομαδιαίας διδασκαλίας (*getWH()*).

Με παρόμοια λογική γράφουμε και τις:

```
void Student::delete1Course( const Course& oneCourse )
{
   int ndx( findNdx(oneCourse.getCode()) );
   if ( ndx >= 0 )  // υπάρχει
   {
      erase1Course( ndx );
      sWH -= oneCourse.getWH();
   }
} // Student::delete1Course

void Student::erase1Course( int ndx )
{
   sCourses[ndx] = sCourses[sNoOfCourses-1];
   --sNoOfCourses;
} // Student::erase1Course
```

Prj04.5.4 Φύλαξη και Φόρτωση

Για τη φύλαξη ενός αντικειμένου *Student* γράφουμε τη μέθοδο:

```
void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
   bout.write( sFirstname, sizeof(sFirstname) );
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
   bout.write( reinterpret_cast<const char*>(&sNoOfCourses),
               sizeof(sNoOfCourses) );
   for ( int k(0); k < sNoOfCourses; ++k )
      bout.write( sCourses[k].s, Course::cCodeSz );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save
```

που διαφέρει από αυτήν της αρχικής εκδοχής στο ότι εδώ υπάρχει η **for** που φυλάγει τους κωδικούς των μαθημάτων.

Η *load()* διαφέρει πιο πολύ από την αρχική.

```
void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      bin.read( reinterpret_cast<char*>(&tmp.sWH), sizeof(sWH) );
      bin.read( reinterpret_cast<char*>(&tmp.sNoOfCourses),
                sizeof(sNoOfCourses) );
      if ( tmp.sNoOfCourses >= tmp.sReserved )
      {
         delete[] tmp.sCourses;
         try
         {
            tmp.sCourses =
                    new Course::CourseKey[((tmp.sNoOfCourses/sIncr)+1)*sIncr];
            tmp.sReserved = ((tmp.sNoOfCourses/sIncr)+1)*sIncr;
         }
         catch( bad_alloc )
         {
            throw StudentXptn( tmp.sIdNum, "load", StudentXptn::allocFailed );
         }
      }
      for ( int k(0); k < tmp.sNoOfCourses; ++k )
         bin.read( tmp.sCourses[k].s, Course::cCodeSz );
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load
```

Εδώ κάνουμε τη φόρτωση με κάπως διαφορετικό τρόπο από αυτόν που χρησιμοποιήσαμε στην *CourseCollection*. Και στις δύο περιπτώσεις αποθηκεύουμε αυτά που διαβάζουμε σε ένα τοπικό αντικείμενο (*tmp*) και τελικώς ανταλλάσσουμε την τιμή του με το `*this`:

- Εκεί, διαβάζαμε τα αντικείμενα τύπου *Course* και τα βάζαμε στον πίνακα με την *insert1Course()*. Η μέθοδος αυτή μετράει τα στοιχεία του πίνακα (αυξάνει τον *sNoOfCourses*). Το πλήθος των στοιχείων που διαβάστηκε από το αρχείο αποθηκεύεται σε μια τοπική μεταβλητή *n*. Ακόμη, η *insert1Course()*, παίρνει και τη δυναμική μνήμη όταν χρειαστεί.
- Εδώ, το πλήθος των στοιχείων που διαβάζεται από το αρχείο αποθηκεύεται στο μέλος του αντικειμένου *tmp*`.`*sNoOfCourses*. Αν χρειάζεται, παίρνουμε την απαραίτητη δυναμι-

κή μνήμη, αφού πρώτα ανακυκλώσουμε όση μνήμη είχε ήδη δεσμεύσει το αντικείμενο. Στη συνέχεια αυτά που διαβάζουμε απόοθηκεύονται κατ' αυθείαν στα στοιχεία του πίνακα.

Εκτός από τις *save()* και *load()*, στην πρώτη εκδοχή της κλάσης, είχαμε και μια άλλη μέθοδο για ανάγνωση από αρχείο, τη *readFromText()*. Θα την κρατήσουμε και εδώ αλλά θα τις δώσουμε πιο «ρεαλιστικό» όνομα: *readPartFromText()*. Θα της κάνουμε όμως και μια άλλη αλλαγή: δεν θα αλλάζει την τιμή του αντικειμένου σε περίπτωση που θα ρίξει εξαίρεση:

```
void Student::readPartFromText( istream& tin )
{
   string line;
   getline( tin, line, '\n' );
   if ( !tin.eof() )
   {
      Student tmp;
      size_t t1Pos( line.find("\t") );
      if ( t1Pos >= line.length() )
         throw StudentXptn( sIdNum, "readPartFromText",
                            StudentXptn::incomplete );
      string str1( line.substr(0, t1Pos) );
      int    iStr1( atoi(str1.c_str()) );
'     try {  tmp.setIdNum( iStr1 );  }
      catch( StudentXptn& x )
      {  strcpy( x.funcName, "readPartFromText" );
         throw;  }
      size_t t2Pos( line.find("\t", t1Pos+1) );
      if ( t2Pos >= line.length() )
         throw StudentXptn( tmp.sIdNum, "readPartFromText",
                            StudentXptn::incomplete );
      tmp.setSurname( line.substr(t1Pos+1, t2Pos-t1Pos-1) );
      tmp.setFirstname( line.substr(t2Pos+1) );
      swap( tmp );
   } // if ( !tin.eof . . .
} // Student::readPartFromText
```

Prj04.5.5 Η Κλάση *Student*

Να πώς έγινε τώρα η κλάση *Student*:

```
class Student
{ // version 2
public:
// constructors, destructor
   explicit Student( int aIdNum=0 );
   Student( const Student& rhs );
   ~Student() {  delete[] sCourses;  };
// copy assignement
   Student& operator=( const Student& rhs );
// getters
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
   unsigned int getWH() const {  return sWH;  }
   unsigned int getNoOfCourses() const { return sNoOfCourses; }
   const Course::CourseKey* getCourses() const
                                          {  return sCourses;  }
// setters
   void setIdNum( int aIdNum );
   void setSurname( string aSurname );
   void setFirstname( string aFirstname );
   void clearCourses() {  sNoOfCourses = 0;  sWH = 0;  }
// 1 Course methods
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
   void add1Course( const Course& oneCourse );
```

```
   void delete1Course( const Course& oneCourse );
// other methods
   void swap( Student& rhs );
   void readPartFromText( istream& tin );
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   enum { sNameSz = 20 };
   enum { sIncr = 3 };
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
   unsigned int       sReserved;

   unsigned int countTabs( string aLine );
   int findNdx( const string& code ) const;
   void insert1Course( const Course::CourseKey& aCode );
   void erase1Course( int ndx );
}; // Student
```

Από την προηγούμενη εκδοχή έχουμε και τις

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

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

Η κλάση εξαιρέσεων είναι:

```
struct StudentXptn
{ // version 2
   enum { allocFailed, outOfRange, incomplete, negIdNum, nonPosIncr,
          fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, const char* sv="" )
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentXptn( int obk, const char* mn, int ec, int ev )
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Prj04.6 Το «Μητρώο Φοιτητών»

Παίρνουμε τα αρχεία `CourseCollection.h` και `CourseCollection.cpp` και με τη *"find-and-replace"* του κειμενογράφου αντικαθιστούμε το "`Course`" με "`Student`". Αυτό που παίρνουμε είναι –σχεδόν– η κλάση *StudentCollection*. Κάθε αντικείμενό της είναι ένα σύνολο αντικειμενων *Student* υλοποιημένο με δυναμικό πίνακα. Μετά από μερικές διορθώσεις έχουμε:

```
class StudentCollection // version 1
{
public:
   StudentCollection();
   ~StudentCollection()  {  delete[] scArr;  };
// getters
   size_t getNOfStudents() const {  return scNOfStudents;  }
   const Student* getArr() const {  return scArr;  }
```

```
// 1 Student
   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) >= 0 );  };
   void delete1Student( int aIdNum );
   void add1Student( const Student& aStudent );
   const Student& get1Student( int aIdNum ) const;
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swap( StudentCollection& rhs );
private:
   enum { scIncr = 30 };
   Student* scArr;
   size_t   scNOfStudents;
   size_t   scReserved;

   StudentCollection( const StudentCollection& rhs ) { };
   StudentCollection& operator=( const StudentCollection& rhs ){};
   void erase1Student( int ndx );
   void insert1Student( const Student& aStudent );
   int findNdx( int aIdNum ) const;
}; // StudentCollection
```

Όπως στην *CourseCollection* έτσι και εδώ, προσπαθούμε να δυσκολέψουμε τη δημιουργία αντιγράφων (πόσα μητρώα φοιτητών θα υπάρχουν;) αχρηστεύοντας και «κρύβοντας» δημιουργό αντιγραφής και τελεστή εκχώρησης.

Θα εστιάσουμε την προσοχή μας στις δύο μεθόδους που έχουν διαφορές από τις αντίστοιχες της *CourseCollection*: τη *delete1Student()* και την *add1Student()*. Η πρώτη είναι απλή:

```
void StudentCollection::delete1Student( int aIdNum )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx >= 0 )  // υπάρχει
   {
      if ( scArr[ndx].getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student
```

αφού αρνούμαστε να συζητήσουμε διαγραφή στην περίπτωση που ο φοιτητής είναι γραμμένος σε μαθήματα· όπως στην *CourseCollection::delete1Course()*, αρνούμαστε να διαγράψουμε ένα μάθημα στο οποίο είναι γραμμένοι φοιτητές. Αργότερα θα δούμε πώς μπορείς να γράψεις μια πληρέστερη μέθοδο.

Η δεύτερη έχει κληρονομήσει ένα πρόβλημα από τη *Student*· το πρόβλημα της *insert1Course()* που δεν αντιμετώπισε ούτε η *add1Course()*: «*Δεν ελέγχει αν υπάρχει στον πίνακα μαθημάτων μάθημα με τέτοιον κωδικό (ακεραιότητα αναφοράς).*» Τώρα όμως θα πρέπει να το λύσουμε: δεν θα πρέπει να εισάγουμε στο «μητρώο φοιτητών» αντικείμενα αν δεν σιγουρεύουμε ότι όλοι οι κωδικοί μαθημάτων αντιστοιχούν σε μαθήματα του πίνακα μαθημάτων (αν μας δίδεται).

Τι θα κάνουμε; Να βάλουμε τον πίνακα μαθημάτων μέσα στη *StudentCollection*; Όχι! Ο πίνακας μαθημάτων είναι αυθύπαρκτος και χρήσιμος και για άλλες δουλειές. Μια σαφώς καλύτερη λύση είναι να βάλουμε ένα βέλος προς τον πίνακα μαθημάτων:

```
   CourseCollection* scPAllCourses;
```

που του δίνουμε αρχική τιμή στον δημιουργό:

```
StudentCollection::StudentCollection()
{
   try
   {
      scReserved = scIncr;
```

```
      scArr = new Student[ scReserved ];
      scNOfStudents = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentCollectionXptn( "StudentCollection",
                                   StudentCollectionXptn::allocFailed);
   }
   scPAllCourses = 0;
} // StudentCollection::StudentCollection
```

Για να μπορούμε να χειριστούμε την τιμή του βέλους γράφουμε μεθόδους:

```
CourseCollection* getPAllCourses() const {  return scPAllCourses;  }
void setPAllCourses( CourseCollection* pCourses )
{  scPAllCourses = pCourses;  }
```

Μετά απο'αυτά, αν στο πρόγραμμά μας έχουμε δηλώσει:

```
CourseCollection allCourses;
StudentCollection allStudents;
```

μπορούμε να δώσουμε:

```
allStudents.setPAllCourses( &allCourses );
```

Ας γράψουμε τώρα την *add1Student()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllCourses != 0 )
      {
         const Course::CourseKey* aStCourses( aStudent.getCourses() );
         for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
         {
           if ( !scPAllCourses->find1Course(aStCourses[k].s) )
              throw StudentCollectionXptn( "add1Student",
                                           StudentCollectionXptn::ref,
                                           aStCourses[k].s );
         }
      }
      insert1Student( aStudent );
   }
} // StudentCollection::add1Student
```

Τα βασικά σημεία:

- Ο έλεγχος για την μοναδικότητα του κλειδιού (αντικειμένου) γίνεται οπωσδήποτε (`findNdx(aStudent.getIdNum()) < 0`).
- Ο έλεγχος των κωδικών των μαθημάτων γίνεται μόνο στην περίπτωση που έχουμε «συνδέσει» πίνακα μαθημάτων (`scPAllCourses != 0`).
- Ενώ διασχίζουμε με τη **for** τον πίνακα κωδικών μαθημάτων aStCourses θα σταματήσουμε μόλις (αν) βρούμε τον πρώτο κωδικό που δεν υπάρχει στον πίνακα μαθημάτων.

Όπως στην *CourseCollection* έχουμε χάσει την πρόσβαση προς τις μεθόδους των στοιχείων (τύπου *Course*) του πίνακα έτσι και εδώ έχουμε χάσει την πρόσβαση προς τις μεθόδους των στοιχείων (τύπου *Student*) του πίνακα. Για να την ανακτήσουμε θα πρέπει να γράψουμε ενδιάμεσες συναρτήσεις. Τέτοιες συναρτήσεις θα πρέπει να γράψουμε τουλάχιστον για τις *add1Course()* και *delete1Course()*. Εδώ όμως χρειάζεται προσοχή: για κάθε στοιχείο του πίνακα sCourses υπάρχει ένα στοιχείο στον *Πίνακα Δηλώσεων Μαθημάτων*· έχουμε δηλαδή *πλεονασμό*. Και ένα πρόβλημα του πλεονασμού είναι το εξής: οι ενημερώσεις (εισαγωγές – διαγραφές) στους δύο πίνακες πρέπει να γίνονται με συνέπεια. Οι *add1Course()* και *delete1Course()* της *Student* δεν έχουν αυτην την ιδιότητα. Θα επανέλθουμε...

Prj04.6.1 Φύλαξη, Φόρτωση και Ευρετήριο

Κατ'αρχήν θα μπορούσαμε να πάρουμε τις *save()* και *load()* από αυτές της *CourseCollection* με «*find-and-replace*» αλλά μια νέα απαίτηση αυτής της δεύτερης εκδοχής του προβλήματος είναι:

> «*Για να ανακτήσουμε τη δυνατότητα κατ' ευθείαν πρόσβασης στα αντικείμενα του αρχείου θα χρειαστούμε ένα* **ευρετήριο** *(index) με στοιχεία τύπου*

```
struct IndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
}; // IndexEntry
```

> *Το πρόγραμμά μας θα πρέπει να οργανώνει αυτά τα στοιχεία σε έναν πίνακα που θα τον φυλάγει σε ένα (μη-μορφοποιημένο) αρχείο* `students.ndx`.»

Να υλοποιήσουμε τα παραπάνω *μέσα* στη *StudentCollection*; Όχι! Ένα αντικείμενο της *StudentCollection* είναι ένα σύνολο που υλοποιείται στη μνήμη. Ο index είναι εργαλείο για τον χειρισμό μιας *StudentCollection* που έχει φυλαχθεί σε αρχείο.

Ο index δεν θα ανήκει σε αντικείμενο και θα έρχεται ως παράμετρος στη *save()*:

```
void StudentCollection::save( ofstream& bout, IndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      index[k].sIdNum = scArr[k].getIdNum();
      index[k].loc = bout.tellp();
      scArr[k].save( bout );
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save
```

Όταν έρχεται η σειρά για φύλαξη στο αρχείο του `scArr[k]` φυλάγουμε στο αντίστοιχο στοιχείο του ευρετηρίου τον αριθμό μητρώου (*sIdNum*) και τη θέση στο αρχείο (*loc*) όπου θα αρχίσει η αποθήκευση του `scArr[k]`. Έτσι, όταν τελειώσει η φύλαξη του `scArr` ο `index` «ξέρει» που αρχίζει η αποθήκευση του κάθε αντικειμένου-στοιχείου του `scArr`.

Η υπόλοιπη διαχείριση του `index` είναι υποχρέωση του προγράμματος-πελάτη της κλάσης.

Για τη φόρτωση δεν χρειάζεται το ευρετήριο:

```
void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(scNOfStudents) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.insert1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
```

```
} // StudentCollection::load
```

Τι είναι η *swapArr()*; Μια μέθοδος που αντιμεταθέτει τους πίνακες δύο αντικειμένων *StudentCollection*:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   std::swap( scArr, rhs.scArr );
   std::swap( scNOfStudents, rhs.scNOfStudents );
   std::swap( scReserved, rhs.scReserved );
} // StudentCollection::swapArr
```

Και γιατί δεν γράφουμε μια *swap()* όπως όλοι οι «κανονικοί» άνθρωποι (προγραμματιστές); Διότι μια τέτοια μέθοδος θα πρέπει να αντιμεταθέτει και τις τιμές των *scPAllCourses* πράγμα όχι και τόσο βολικό.[6]

Prj04.7 Η Κλάση *StudentInCourse*

Θα ξαναγράψουμε τη *StudentInCourse* με βάση τη συνταγή μας και παίρνοντας υπόψη μας τις αλλαγές που κάναμε στην *Course*:

```
class StudentInCourse
{
public:
// . . .
private:
   unsigned int      sicSIdNum;  // αριθμός μητρώου
   Course::CourseKey sicCCode;   // κωδικός μαθήματος
   float             sicMark;    // βαθμός στο μάθημα
}; // StudentInCourse
```

Δύο αντικείμενα αυτής της κλάσης, *a* και *b*, είναι ίσα αν και μόνον αν αναφέρονται στον ίδιο φοιτητή (*a.sicSIdNum* == *b.sicSIdNum*) για το ίδιο μάθημα (*a.sicCCode* == *b.sicCCode*). Δηλαδή, το ζεύγος (*sicSIdNum, sicCCode*) είναι *κλειδί*. Για χρήση στην κλάση εξαιρέσεων (αλλά και αλλού) ορίζουμε στην περιοχή "**public**" της κλάσης τον τύπο:

```
   struct SICKey
   {
      unsigned int      sIdNum;
      Course::CourseKey CCode;
      explicit SICKey( int aIdNum=0, string aCCode="" )
      {  sIdum = aIdNum;  CCode = Course::CourseKey(aCCode);  }
   }; // SICKey
```

και τη

```
   SICKey getKey() const
   {  return SICKey( sicSIdNum, sicCCode.s );  }
```

Για τον τύπο αυτόν μπορούμε να επιφορτώσουμε τον "`!=`":

```
bool operator!=( const StudentInCourse::SICKey& lhs,
                 const StudentInCourse::SICKey& rhs )
{  return ( (lhs.sIdNum != rhs.sIdNum) || (lhs.CCode != rhs.CCode) );  }
```

και στη συνέχεια για τον *StudentInCourse*:

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

Ας εφαρμόσουμε τώρα τη συνταγή μας.

ΑΝ: Η αναλλοίωτη της κλάσης θα είναι:

$$0 <= sicSIdNum \;\&\&$$
$$((strlen(sicCCode.s) == Course::cCodeSz\text{-}1) \;||\; (sicCCode.s) == 0)) \;\&\&$$
$$0 <= sicMark <= 10$$

[6] Κάτι δεν πάει καλά με τη σχεδίαση…

ΔΕ: Ο ερήμην δημιουργός (με αρχική τιμή) δηλώνεται ως:

```
   StudentInCourse( int aIdNum=0, string aCCode="" );
```

και ορίζεται:

```
StudentInCourse::StudentInCourse( int aIdNum, string aCCode )
{
   if ( aIdNum < 0 )
      throw StudentInCourseXptn( SICKey(), "StudentInCourse",
                                 StudentInCourseXptn::negIdNum, aIdNum );
   sicSIdNum = aIdNum;
   if ( aCCode.length() != Course::cCodeSz-1 &&
        aCCode.length() != 0 )
      throw StudentInCourseXptn( SICKey(sicSIdNum), "StudentInCourse",
                                 StudentInCourseXptn::keyLen, aCCode.c_str());
   sicCCode = Course::CourseKey( aCCode );
   sicMark = 0;
} // StudentInCourse::StudentInCourse
```

ΔΑ, ΤΕ, ΚΑ: Ο δημιουργός αντιγραφής και ο τελεστής εκχώρησης που μας δίνει ο μεταγλωττιστής κάνουν σωστά τις δουλειές τους. Πάντως, κατά τη συνήθειά μας, θα γράψουμε έναν κενό καταστροφέα:

```
   ~StudentInCourse() { };
```

GE: Οι μέθοδοι "*get*", εκτός από τη *getKey()* που είδαμε παραπάνω:

```
   unsigned int getSIdNum() const {  return sicSIdNum;  }
   const char* getCCode() const {  return sicCCode.s;  }
   float getMark() const {  return sicMark;  }
```

SE: Και οι μέθοδοι "*set*":

```
void StudentInCourse::setSIdNum( int aIdNum )
{
   if ( aIdNum <= 0 )
      throw StudentInCourseXptn( getKey(), "setSIdNum",
                                 StudentInCourseXptn::negIdNum, aIdNum );
   sicSIdNum = aIdNum;
} // StudentInCourse::setSIdNum

void StudentInCourse::setCCode( string aCCode)
{
   if ( aCCode.length() != Course::cCodeSz-1 )
      throw StudentInCourseXptn( getKey(), "setCCode",
                                 StudentInCourseXptn::keyLen, aCCode.c_str());
   sicCCode = Course::CourseKey( aCCode );
} // StudentInCourse::setCCode

void StudentInCourse::setMark( float aMark )
{
   if ( aMark < 0 || 10 < aMark )
      throw StudentInCourseXptn( getKey(), "setMark",
                                 StudentInCourseXptn::rangeError, aMark );
   sicMark = aMark;
} // StudentInCourse::setMark
```

Τη *setIdNumCCode()* που είχαμε θα την κρατήσουμε; Όχι, διότι, όπως πιθανότατα μαντεύεις, στη συνέχεια θα γράψουμε και μια κλάση *StudentInCourseCollection* όπου και θα γίνονται αυτά που έκανε η *setIdNumCCode()*.

Άλλες Μέθοδοι: Η *save()* είναι αυτή της αρχικής εκδοχής με μικρές προσαρμογές στις εξαιρέσεις και στη φύλαξη του κωδικού μαθήματος:

```
void StudentInCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentInCourseXptn( getKey(), "save",
                                 StudentInCourseXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sicSIdNum), sizeof(sicSIdNum) );
   bout.write( sicCCode.s, Course::cCodeSz ); // κωδικός μαθήματος
```

```
   bout.write( reinterpret_cast<const char*>(&sicMark), sizeof(sicMark) );
   if ( bout.fail() )
      throw StudentInCourseXptn( getKey(), "save",
                                 StudentInCourseXptn::cannotWrite );
} // StudentInCourse::save
```

Για τη δεύτερη εφαρμογή θα χρειαστούμε και μια *load()*:

```
void StudentInCourse::load( istream& bin )
{
   StudentInCourse tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sicSIdNum), sizeof(sicSIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sicCCode.s, Course::cCodeSz );        // κωδικός μαθήματος
      bin.read( reinterpret_cast<char*>(&tmp.sicMark), sizeof(sicMark) );
      if ( bin.fail() )
         throw StudentInCourseXptn( getKey(), "load",
                                    StudentInCourseXptn::cannotWrite );
      *this = tmp;
   }
} // StudentInCourse::load
```

Η κλάση θα είναι:

```
class StudentInCourse
{
public:
   struct SICKey
   {
      unsigned int      sIdNum;
      Course::CourseKey CCode;
      explicit SICKey( int aIdNum=0, string aCCode="" )
      {  sIdNum = aIdNum;  CCode = Course::CourseKey(aCCode);  }
   }; // SICKey
   explicit StudentInCourse( int aIdNum=0, string aCCode="" );
   ~StudentInCourse() { };
// getters
   unsigned int getSIdNum() const {  return sicSIdNum;  }
   const char* getCCode() const {  return sicCCode.s;  }
   float getMark() const {  return sicMark;  }
   SICKey getKey() const
   {  return SICKey( sicSIdNum, sicCCode.s );  }
// setters
   void setSIdNum( int aIdNum );
   void setCCode( string aCode);
   void setMark( float aMark );
// other
   void save( ostream& bout ) const;
   void load( istream& bin );
private:
   unsigned int      sicSIdNum;  // αριθμός μητρώου
   Course::CourseKey sicCCode;   // κωδικός μαθήματος
   float             sicMark;    // βαθμός στο μάθημα
}; // StudentInCourse
```

Ακόμη, έχουμε επιφορτώσει τον "`!=`":

```
bool operator!=( const StudentInCourse::SICKey& lhs,
                 const StudentInCourse::SICKey& rhs )
{  return ( (lhs.sIdNum != rhs.sIdNum) || (lhs.CCode != rhs.CCode) );  }

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

Και η κλάση εξαιρέσεων:

```
struct StudentInCourseXptn
{
   enum { negIdNum, keyLen, rangeError, unknownCCode,
          fileNotOpen, cannotWrite };
   StudentInCourse::SICKey objKey;
```

```
   char                    funcName[100];
   int                     errorCode;
   char                    errStrVal[100];
   float                   errFltVal;
   StudentInCourseXptn( const StudentInCourse::SICKey obk,
                        const char* mn, int ec, const char* sv="" )
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentInCourseXptn( const StudentInCourse::SICKey& obk,
                        const char* mn, int ec, float fv )
      :  objKey( obk ),  errorCode( ec ),  errFltVal( fv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentInCourseXptn
```

Prj04.8 Η Κλάση *StudentInCourseCollection*

Αν θυμηθούμε πώς γράψαμε τις *CourseCollection* και *StudentCollection* και πάρουμε υπόψη μας ότι «*το ζεύγος (sicSIdNum, sicCCode) είναι κλειδί*» γράφουμε μια πρώτη μορφή της

```
class StudentInCourseCollection
{
public:
   StudentInCourseCollection();
   ~StudentInCourseCollection()  {  delete[] siccArr;  };
// getters
   size_t getNOfStudentInCourses() const
   {  return siccNOfStudentInCourses;  }
   const StudentInCourse* getArr() const {  return siccArr;  }
// 1 StudentInCourse
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) >= 0 );  }
   const StudentInCourse& get1StudentInCourse( int aIdNum,
                                               string code ) const;
   void delete1StudentInCourse( int aIdNum, string code );
   void add1StudentInCourse( const StudentInCourse& aStdnInCrs );
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swapArr( StudentInCourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { siccIncr = 50 };
   StudentInCourse* siccArr;
   size_t           siccNOfStudentInCourses;
   size_t           siccReserved;

   StudentInCourseCollection( const StudentInCourseCollection& rhs ) { };
   StudentInCourseCollection& operator=(
                                  const StudentInCourseCollection& rhs ) { };

   void erase1StudentInCourse( int ndx );
   void insert1StudentInCourse( const StudentInCourse& aStdnInCrs );
   int findNdx( int aIdNum, string code ) const;
}; // StudentInCourseCollection
```

Ενδεικτικώς ας δούμε τη

```
int StudentInCourseCollection::findNdx( int aIdNum, string code ) const
{
   int ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearch( siccArr, siccNOfStudentInCourses,
                       0, siccNOfStudentInCourses-1,
                       StudentInCourse(aIdNum,code) );
```

```
   return ndx;
} // StudentInCourseCollection::findNdx
```

Εδώ πρόσεξε τα εξής:

- Αφού το κλειδί για τα αντικείμενα τύπου *StudentInCourse* είναι (*sicSIdNum*, *sicCCode*) η μέθοδος πρέπει να τροφοδοτηθεί με τα αντίστοιχα *aIdNum* και *code*. Τις ίδιες παραμέτρους έχουν και οι *find1StudentInCourse()*, *delete1StudentInCourse()*, *get1StudentInCourse()* και *delete1StudentInCourse()*.
- Πριν καλέσουμε τη *linSearch()* ελέγχουμε και τα δύο τμήματα του κλειδιού· θα πρέπει να έχουμε: *aIdNum* > 0 && *code*.*length*() == *Course*::*cCodeSz*-1.

Φυσικά, το ενδιαφέρον βρίσκεται στις *delete1StudentInCourse()* και *add1StudentInCourse()*. Η δεύτερη θα παίξει (και) τον ρόλο της *setIdNumCCode()* του Project 3.

Για να μπορεί η *add1StudentInCourse()* να παίξει τον ρόλο της *setIdNumCCode()* θα πρέπει να έχει πρόσβαση στον πίνακα μαθημάτων και στο μητρώο φοιτητών· θα βάλουμε και εδώ παραμέτρους για τους πίνακες; Όχι! Όπως κάναμε στη *StudentCollection* θα βάλουμε:

```
private:
// . . .
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;
```

με αρχικές τιμές στον δημιουργό:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   try
   {
      siccReserved = siccIncr;
      siccArr = new StudentInCourse[ siccReserved ];
      siccNOfStudentInCourses = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentInCourseCollectionXptn( "StudentInCourseCollection",
                                 StudentInCourseCollectionXptn::allocFailed );
   }
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection
```

και –για τον χειρισμό τους– τις

```
public:
// . . .
   const StudentCollection* getPAllStudents() const
   {  return siccPAllStudents;  }
   const CourseCollection* getPAllCourses( ) const
   {  return siccPAllCourses;  }
// setters
   void setPAllStudents( const StudentCollection* pStudents )
   {  siccPAllStudents = pStudents;  }
   void setPAllCourses( const CourseCollection* pCourses )
   {  siccPAllCourses = pCourses;  }
```

Μετά απο΄αυτά, αν στο πρόγραμμά μας έχουμε δηλώσει:

```
   CourseCollection allCourses;
   StudentCollection allStudents;
   StudentInCourseCollection allEnrollments;
```

μπορούμε να δώσουμε:

```
   allEnrollments.setPAllStudents( &allStudents );
   allEnrollments.setPAllCourses( &allCourses );
```

Η *delete1StudentInCourse()* και *add1StudentInCourse()* θα κάνουν ελέγχους –και ενημερώσεις– αν βρίσκουν `siccPAllStudents != 0` ή/και `siccPAllCourses != 0` για να ανταποκριθούν στην απαίτηση: «*οι ενημερώσεις (εισαγωγές – διαγραφές) στους δύο πίνακες*

[sCourses της *Student* και siccArr της *StudentInCourseCollection*] *πρέπει να γίνονται με συνέπεια.*»

Ας δούμε λοιπόν ένα σχέδιο για την *add1StudentInCourse()*:

- Πριν από οποιαδήποτε ενέργεια θα πρέπει να ελέγξουμε αν υπάρχει ήδη καταχωρισμένη η εγγραφή του φοιτητή στο συγκεκριμένο μάθημα. Αν υπάρχει δεν υπάρχει δουλειά για τη μέθοδο:

```
void add1StudentInCourse( const StudentInCourse& aStdnInCrs )
{
   if ( στον siccArr δεν υπάρχει στοιχείο με
                              (aStdnInCrs.getSIdNum(),aStdnInCrs.getCCode()) )
   {
   // . . .
   }
} // StudentInCourseCollection::add1StudentInCourse
```

- Για να προχωρήσουμε στην ενημέρωση του πίνακα (εισαγωγή στοιχείου) θα πρέπει να έχουμε σιγουρέψει ότι πρόκειται για εγγραφή υπαρκτού φοιτητή σε υπαρκτό μάθημα:

```
   if ( siccPAllCourses == 0 )
      throw …
   if ( δεν υπάρχει μάθημα με κωδικό aStdnInCrs.getCCode() )
      throw …
   if ( siccPAllStudents == 0 )
      throw …
   if ( δεν υπάρχει φοιτητής με α.μ. aStdnInCrs.getSIdNum() )
      throw …
```

- Αν βρούμε όλα όσα χρειαζόμαστε κάνουμε τις ενημερώσεις:

```
   Ενημέρωσε το μάθημα (αριθμός φοιτητών)
   Πάρε το αντικείμενο του μαθήματος
   Ενημέρωσε τον φοιτητή (εισαγωγή μαθήματος)
   Κάνε εισαγωγή του aStdnInCrs στον siccArr
```

Για να ενημερώσουμε το μάθημα πρέπει να καλέσουμε την *add1Student()* της συλλογής (*CourseCollection*) που δείχνει το *siccPAllCourses*:

```
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
```

Από την ίδια συλλογή παίρνουμε το αντικείμενο του μαθήματος για να το χρησιμοποιήσουμε στην ενημέρωση το αντικείμενο του φοιτητή:

```
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
```

Για να ενημερώσουμε το αντικείμενο του φοιτητή πρέπει να καλέσουμε την *add1Course* της συλλογής (*StudentCollection*) που δείχνει το *siccPAllStudents*:[7]

```
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
```

Μετά από αυτά εισάγουμε το *aStdnInCrs* στον πίνακα siccArr της συλλογής *StudentInCourseCollection*:

```
      insert1StudentInCourse( aStdnInCrs );
```

Να ολόκληρη η μέθοδος:

```
void StudentInCourseCollection::add1StudentInCourse(
                                          const StudentInCourse& aStdnInCrs )
{
   if ( findNdx(aStdnInCrs.getSIdNum(), aStdnInCrs.getCCode()) < 0 )
   { // δεν υπάρχει στοιχείο
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                     StudentInCourseCollectionXptn::noCrs );
```

[7] Δεν είναι απαραίτητο να χρησιμοποιήσουμε την τοπική μεταβλητή *oneCourse*· θα μπορούσαμε να γράψουμε:

```
      pAllStudents->add1Course( aStdnInCrs.getSIdNum(),
                        pAllCourses->get1Course( aStdnInCrs.getCCode() );
```

```
      if ( !(siccPAllCourses->find1Course(aStdnInCrs.getCCode())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                StudentInCourseCollectionXptn::unknownCrs,
                                          aStdnInCrs.getCCode() );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                 StudentInCourseCollectionXptn::noStdnt );
      if ( !(siccPAllStudents->find1Student(aStdnInCrs.getSIdNum())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                               StudentInCourseCollectionXptn::unknownStdn,
                                          aStdnInCrs.getSIdNum() );
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
      insert1StudentInCourse( aStdnInCrs );
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Η *delete1StudentInCourse()* είναι κάπως πιο απλή:

```
void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                        string code )
{
   int ndx( findNdx( aIdNum, code );
   if ( ndx >= 0 ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                   StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                 StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course(code) );
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      erase1StudentInCourse( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse
```

Η *StudentInCourseCollection* γίνεται τελικώς:

```
class CourseCollection;
class StudentCollection;

class StudentInCourseCollection
{ // version 1
public:
   StudentInCourseCollection();
   ~StudentInCourseCollection()  {  delete[] siccArr;  };
// getters
   size_t getNOfStudentInCourses() const
   {  return siccNOfStudentInCourses;  }
   const StudentInCourse* getArr() const {  return siccArr;  }
   const StudentCollection* getPAllStudents() const
   {  return siccPAllStudents;  }
   const CourseCollection* getPAllCourses( ) const
   {  return siccPAllCourses;  }
// setters
   void setPAllStudents( StudentCollection* pStudents )
   {  siccPAllStudents = pStudents;  }
   void setPAllCourses( CourseCollection* pCourses )
   {  siccPAllCourses = pCourses;  }
// 1 StudentInCourse
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) >= 0 );  }
   void delete1StudentInCourse( int aIdNum, string code );
   void add1StudentInCourse( const StudentInCourse& aStdnInCrs );
   const StudentInCourse& get1StudentInCourse( int aIdNum,
                                               string code ) const;
```

```
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swapArr( StudentInCourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { siccIncr = 50 };
   StudentInCourse*   siccArr;
   size_t             siccNOfStudentInCourses;
   size_t             siccReserved;
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;

   StudentInCourseCollection( const StudentInCourseCollection& rhs ) { };
   StudentInCourseCollection& operator=(
                                  const StudentInCourseCollection& rhs ) { };
   void erase1StudentInCourse( int ndx );
   void insert1StudentInCourse( const StudentInCourse& aStdnInCrs );
   int findNdx( int aIdNum, string code ) const;
}; // StudentInCourseCollection
```

Πρόσεξε τις δύο *προειδοποιήσεις δήλωσης* για τις κλάσεις *CourseCollection* και *StudentCollection*. Είναι απαραίτητες για να γίνουν δεκτές από τον μεταγλωττιστή οι δηλώσεις

```
   StudentCollection* siccPAllStudents;
   CourseCollection*  siccPAllCourses;
```

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentInCourseCollectionXptn
{ // version 1
   enum { allocFailed, notFound, noCrs, unknownCrs, noStdnt,
          unknownStdn, fileNotOpen, cannotWrite, cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   char errIntVal;
   StudentInCourse::SICKey errSICVal;
   StudentInCourseCollectionXptn( const char* mn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
   StudentInCourseCollectionXptn( const char* mn, int ec, int iv )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
   StudentInCourseCollectionXptn( const char* mn, int ec,
                              const StudentInCourse::SICKey& sk )
      :  errorCode( ec ),  errSICVal( sk )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
}; // StudentInCourseCollectionXptn
```

Prj04.9 Πώς θα Γίνονται οι Ενημερώσεις

Ας πούμε ότι θέλουμε να κάνουμε εισαγωγή της εγγραφής ενός φοιτητή σε κάποιο μάθημα· θα πρέπει

- Να ενημερώσουμε ένα μέλος του πίνακα μαθημάτων (τύπου *Course Collection*), με την *add1Student()*.
- Να ενημερώσουμε ένα μέλος του μητρώου φοιτητών (τύπου *StudentCollection*), με την *add1Course()*.
- Να εισαγάγουμε ένα νέο στοιχείο στον πίνακα εγγραφών στα μαθήματα (τύπου *StudentInCourseCollection*) με την *add1StudentInCourse()*.

Ποιος διασφαλίζει ότι όλα αυτά θα γίνουν σωστά ώστε να είναι σωστό το περιεχόμενο των πινάκων μας; Το πρόγραμμα που τους χρησιμοποιεί; Ούτε για αστείο! Αυτό πρέπει να διασφαλίζεται από αυτόν που γράφει τις κλάσεις. *Εδώ* θα γίνει.

Η *StudentInCourseCollection::add1StudentInCourse()*, όπως τη γράψαμε, κάνει όλες τις ενημερώσεις που παραθέτουμε παραπάνω. Αυτό όμως δεν διασφαλίζει οτιδήποτε, αφού ο προγραμματιστής μπορεί να βάλει κλήσεις (και) προς τις άλλες μεθόδους. Πώς διορθώνεται αυτό; Έχουμε δύο επιλογές:

- Να γράψουμε τις (*Course Collection*::)*add1Student()*, (*StudentCollection*::)*add1Course()* έτσι ώστε να κάνουν όλες τις ενημερώσεις. Με αυτήν την επιλογή θα έχεις το δικαίωμα να κάνεις εισαγωγή εγγραφής σε ένα μάθημα με οποιαδήποτε από τις τρεις μεθόδους. Αλλά και κάθε φορά που θα αλλάζεις το λογισμικό σου θα πρέπει να αλλάζεις και τις τρεις μεθόδους με συνεπή τρόπο.
- Να κρύψουμε (σε περιοχή **private**) τις (*CourseCollection*::)*add1Student()*, (*StudentCollection*::)*add1Course()* ώστε η *add1StudentInCourse()* να είναι η μόνη υπεύθυνη για όλες τις ενημερώσεις.

Η δεύτερη επιλογή έχει αυτό το πολύ σημαντικό πλεονέκτημα: μια μεθοδος «*είναι η μόνη υπεύθυνη για όλες τις ενημερώσεις.*» Η πρώτη επιλογή, χωρίς αυτό το «προσόν», σου εγγυάται(!) ότι –συν τω χρόνω, με τις αλλαγές και προσαρμογές που θα πρέπει να γίνονται– θα έχεις λογισμικό και δεδομένα με πολλά προβλήματα.

Φυσικά, θα πρέπει να έχουμε και *μια μονον μέθοδο υπεύθυνη για τη όλες τις ενημερώσεις που αφορούν τη διαγραφή* ενός φοιτητή από ένα μάθημα.

Πώς υλοποιείται η επιλογή μας;

- Δηλώνουμε στην *CourseCollection* και στην *StudentCollection*:

```
friend void StudentInCourseCollection::add1StudentInCourse(
                                        const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                               string code );
```

- «Κρύβουμε» στην περιοχή **private** της *CourseCollection* τις *add1Student* και *delete1Student*.
- «Κρύβουμε» στην περιοχή **private** της *StudentCollection* τις *add1Course* και *delete1Course*.

Prj04.10 Οι Άλλες Συλλογές Τελικώς

Τώρα μπορούμε να δώσουμε τις τελικές –αν και όχι πλήρεις– μορφές των κλάσεων *CourseCollection* και *StudentCollection*.

Prj04.10.1 Η Κλάση *CourseCollection*

Θα ξαναδούμε τώρα την *CourseCollection::delete1Course()*. Η λύση που δώσαμε στην §Prj04.3.1 φαίνεται κάπως «τεμπέλικη» διότι αυτό που λέμε στην πραγματικότητα είναι το εξής: Θέλεις να διαγράψεις ένα μάθημα στο οποίο είναι γραμμένοι φοιτητές (*cNoOfStudents* > 0); Τότε:

- Διάγραψε πρώτα μια προς μια τις εγγραφές των φοιτητών στο συγκεκριμένο μάθημα ενημερώνοντας παραλλήλως τα αντικείμενα των φοιτητών.
- Μετά από αυτό (αφού θα έχεις *cNoOfStudents* == 0) διάγραψε και το αντικείμενο του μαθήματος.

Παίρνοντας υπόψη μας και αυτά που λέγαμε στην Prj04.4 σου προτείνουμε (ως άσκηση) το εξής:

- Εφοδίασε το σύστημά σου με έναν πίνακα τύπου *DeletedCourseCollection* και έναν *DeletedStudentInCourseCollection*[8] με στοιχεία αντίστοιχων κλάσεων

```
class DeletedCourse
{
private:
// . . .
public:
   Course dcCourse;
   time_t dcDelTime; // ή DateTime dcDelTime (χρόνος διαγραφής)
}; // DeletedCourse
class DeletedStudentInCourse
{
private:
// . . .
public:
   StudentInCourse dsiccEnroll;
   time_t          dsiccDelTime; // ή DateTime dsiccDelTime
}; // DeletedStudentInCourse
```

- Μην πειράξεις την *CourseCollection::delete1Course()* αλλά γράψε μιαν άλλη μέθοδο, ας την πούμε *CourseCollection::moveOut1Course()*, που θα κάνει τα εξής:
  - Θα διαγράφει τις εγγραφές στο μάθημα από τον πίνακα δηλώσεων και αφού εξοπλίσει την κάθε μια με τον χρόνο διαγραφής (μετατροπή σε αντικείμενο *DeletedStudentInCourse*) θα τις εισάγει στον πίνακα τύπου *DeletedStudentInCourseCollection*. Προφανώς θα χρειαστείς και μια *StudentInCourseCollection::moveOut1StudentInCourse()*.
  - Θα διαγράφει το αντικείμενο του μαθήματος από τον πίνακα μαθημάτων και αφού το μετατρέψει σε αντικείμενο *DeletedCourse* θα το εισάγει στον πίνακα *DeletedCourseCollection*.
  - Φυσικά, θα πρέπει να ενημερώνει και τα αντικείμενα των φοιτητών που έχουν εγγραφεί στο μάθημα που διαγράφεται.

Όλες οι διαγραφές από τον πίνακα δηλώσεων θα πρέπει να γίνονται με τη *StudentInCourseCollection::delete1StudentInCourse()* και μόνον. Για να έχεις πρόσβαση από τον πίνακα μαθημάτων στον πίνακα των εγγραφών στα μαθήματα θα χρειαστείς ένα βέλος:

```
   StudentInCourseCollection* ccPAllEnrollments;
```

που θα παίρνει τιμή "`0`" όταν δημιουργείται η συλλογή (πίνακας μαθημάτων) και θα το διαχειρίζεσαι με τις:

```
const StudentInCourseCollection*
                        CourseCollection::getPAllEnrollments() const
{  return ccPAllEnrollments;  }

void CourseCollection::setPAllEnrollments(
                           StudentInCourseCollection* pEnrollments )
{  ccPAllEnrollments = pEnrollments;  }
```

Αφού σχεδιάσεις και υλοποιήσεις τους πίνακες διαγραφέντων μάλλον θα πρέπει να βάλεις αντίστοιχα εργαλεία και για αυτούς.

Ο ορισμός της κλάσης *CourseCollection* είναι:

```
class CourseCollection
{  // version 1
friend void StudentInCourseCollection::add1StudentInCourse(
                                       const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                               string code );
public:
   CourseCollection();
```

[8] Φυσικά θα χρειαστεί και ένας πίνακας *DeletedStudentCollection*.

```
   ~CourseCollection()  {  delete[] ccArr;  };
// getters
   size_t getNOfCourses() const {  return ccNOfCourses;  }
   const Course* getArr() const {  return ccArr;  }
//   const StudentInCourseCollection* getPAllEnrollments() const
//   {  return ccPAllEnrollments;  }
// setters
//   void setPAllEnrollments( StudentInCourseCollection*
//                                                    pEnrollments)
//   {  ccPAllEnrollments = pEnrollments;  }
// 1 Course
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) >= 0 );  };
   void delete1Course( string code );
   void add1Course( const Course& aCourse );
   const Course& get1Course( string code ) const;
// other
   void save( ofstream& bout ) const;
   void load( ifstream& bin );
   void swap( CourseCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { ccIncr = 30 };
   Course*                    ccArr;
   size_t                     ccNOfCourses;
   size_t                     ccReserved;
//   StudentInCourseCollection* ccPAllEnrollments;

   CourseCollection( const CourseCollection& rhs ) { };
   CourseCollection& operator=( const CourseCollection& rhs ) { };
   void erase1Course( int ndx );
   void insert1Course( const Course& aCourse );
   void add1Student( string code );
   void delete1Student( string code );
   int findNdx( const string& code ) const;
}; // CourseCollection
```

Σε σχόλια έχουμε βάλει τα εργαλεία για την πρόσβαση στον πίνακα εγγραφών στα μαθήματα.[9]

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct CourseCollectionXptn
{  // version 1
   enum { allocFailed, notFound, entity, cannotDel, prereqRef,
          noEnroll, enrollRef, fileNotOpen, cannotWrite,
          cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   int  errIntVal;
   CourseCollectionXptn( const char* mn, int ec,
                         const char* sv="", int iv=0 )
```

[9] Και πώς θα δεχθεί ο μεταγλωττιστής τη δήλωση της *ccPAllEnrollments*; Δεν θα χρειαστεί μια προειδοποίηση δήλωσης της *StudentInCourseCollection* πριν από τη δήλωση της `CourseCollection`; Εδώ έχουμε και άλλες απαιτήσεις από τις δηλώσεις "`friend`" των `StudentInCourseCollection::add1StudentInCourse()` και `StudentInCourseCollection::delete1StudentInCourse()`. Εδώ το πρόβλημα λύνεται με τη δήλωση της *StudentInCourseCollection* πριν από τη δήλωση της *CourseCollection*. Αφού παρόμοια ισχύουν και για την *StudentCollection*, στο πρόγραμμά μας θα πρέπει να βάλουμε:

```
#include "StudentInCourseCollection.h"
#include "CourseCollection.h"
#include "StudentCollection.h"
```

Έτσι, όταν έρχεται η δήλωση της *CourseCollection* (και μετά, της *Student Collection*) ο μεταγλωττιστής έχει ήδη βρει τη *StudentInCourseCollection* και τις μεθόδους της *add1StudentInCourse()* και *delete1StudentInCourse()*.

```
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
}; // CourseCollectionXptn
```

Παρατήρηση: ▶

Κρύψαμε τον δημιουργό αντιγραφής και τον τελεστή εκχώρησης για να εμποδίσουμε την αντιγραφή αντικειμένων κλάσης *CourseCollection*. Τελικώς, απλώς τη δυσκολέψαμε: με τις *getNOfCourses()*, *getArr()* και *add1Course()* μπορείς να δημιουργήσεις αντίγραφα. Πάντως, έχουμε την εξής ελπίδα: ο προγραμματιστής που θα προσπαθήσει να δημιουργήσει αντίγραφα αντικειμένων κλάσης *CourseCollection*, θα δει τα αχρηστευμένα και θα το ξανασκεφτεί...◀

Prj04.10.2 Η Κλάση *StudentCollection*

Το πρώτο πράγμα που θα πρέπει να κάνουμε είναι η διόρθωση της *add1Student()*. Αφού είπαμε ότι η εισαγωγή εγγραφών σε μαθήματα θα γίνεται από «μια πόρτα», την *add1StudentInCourse()*, αυτή θα πρέπει να καλούμε στη **for** της *add1Student()*. Ο έλεγχος

```
if ( !scPAllCourses->find1Course(aStCourses[k].s) )
```

είναι περιττός αφού γίνεται από την *add1StudentInCourse()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      insert1Student( aStudent );
      const Course::CourseKey* aStCourses( aStudent.getCourses() );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                                     StudentInCourse(aStudent.getIdNum(),
                                                     aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Πρόσεξε τα εξής:

- Θα χρειαστούμε ένα βέλος –ας το πούμε *scPAllEnrollments*– προς τον *Πίνακα Δηλώσεων Μαθημάτων*. Η τιμή του ελέγχεται στη δεύτερη **if**. *Οι δηλώσεις και οι μέθοδοι χειρισμού του βέλους προς τον πίνακα μαθημάτων θα αντικατασταθούν από τα αντίστοιχα του scPAllEnrollments*.
- Με την *insert1Student()* εισάγεται και ο πίνακας με τους κωδικούς μαθημάτων. Στη συνέχεια, η *StudentInCourseCollection::add1StudentInCourse()* θα καλεί τη *StudentCollection::add1Course()* που καλεί τη *Student::add1Course()* για να εισαγάγει τον κάθε κωδικό στον πίνακα κωδικών μαθημάτων. Αλλά η τελευταία δεν κάνει εισαγωγή αν τον βρει ήδη στον πίνακα.
- Υπάρχει μια σοβαρή διαφορά της *StudentCollection::add1Student()* από την *CourseCollection::add1Course()*:
  - Στο μητρώο φοιτητών εισάγουμε οποιοδήποτε αντικείμενο κλάσης *Student* χωρίς οποιοδήποτε πρόβλημα. Και αυτό διότι το αντικείμενο *Student*, με τον πίνακα μαθημάτων που περιέχει, μας επιτρέπει να ενημερώσουμε σωστά τον πίνακα δηλώσεων μαθημάτων και τον πίνακα μαθημάτων.
  - Στον πίνακα μαθημάτων εισάγουμε αντικείμενα κλάσης *Course* με *μηδενισμένο το ocNoOfStudents*. Γιατί; Διότι ένα αντικείμενο *Course* έχει *μόνον το πλήθος* των

φοιτητών που το ζήτησαν και *όχι τους αριθμούς μητρώου*. Έτσι είναι αδύνατη η ενημέρωση του πίνακα δηλώσεων μαθημάτων και των αντικειμένων με τα στοιχεία των φοιτητών.

Ας έλθουμε τώρα στη *StudentCollection::delete1Student()*, που την κάναμε «τεμπέλικη», αφού σβήνει μόνον φοιτητές που δεν είναι γραμμένοι σε μαθήματα. Παρ' όλο που η διαγραφή των εγγραφών ενός φοιτητή σε 4 ή 5 μαθήματα από τον *Πίνακα Δηλώσεων Μαθημάτων* δεν είναι τόσο «τραγική» όσο η διαγραφή των εγγραφών μερικών δεκάδων φοιτητών σε ένα μάθημα, θα σου προτείνουμε να χειριστείς τη διαγραφή φοιτητή με παρόμοιο τρόπο:

- Όρισε μια κλάση:

```
class DeletedStudent
{
private:
// . . .
public:
   Student dsStudent;
   time_t  dsDelTime; // ή DateTime dsDelTime (χρόνος διαγραφής)
}; // DeletedStudent
```

- Μην πειράξεις την («τεμπέλικη») *StudentCollection::delete1Student()* αλλά γράψε μιαν άλλη μέθοδο, ας την πούμε *StudentCollection::moveOut1Student()*, που θα κάνει τα εξής:
  - Θα διαγράφει τις εγγραφές του φοιτητή στα μαθήματα από τον πίνακα δηλώσεων και αφού εξοπλίσει την κάθε μια με τον χρόνο διαγραφής (μετατροπή σε αντικείμενο *DeletedStudentInCourse*) θα τις εισάγει στον πίνακα τύπου *DeletedStudentInCourseCollection* που είδαμε πιο πριν.
  - Θα διαγράφει το αντικείμενο του φοιτητή από το μητρώο φοιτητών και αφού το μετατρέψει σε αντικείμενο *DeletedStudent* θα το εισάγει στον πίνακα *DeletedStudentCollection*.
  - Θα πρέπει να ενημερώνει και τα αντικείμενα των μαθημάτων στα οποία είχε εγγραφεί ο φοιτητής που διαγράφεται.

Θα πρέπει ακόμη να γράψουμε τις «ενδιάμεσες» μεθόδους *add1Course()* και *delete1Course()*. Να θυμίσουμε ότι θα είναι *εσωτερικές* και θα καλούντσι μόνον από τις *φίλες StudentInCourseCollection::add1StudentInCourse()* και *StudentInCourseCollection::delete1StudentInCourse()*.

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   scArr[ndx].add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   scArr[ndx].delete1Course( aCourse );
} // StudentCollection::add1Course
```

Να πώς δηλώνεται η κλάση *StudentCollection*:

```
class StudentCollection
{  // version 1
friend void StudentInCourseCollection::add1StudentInCourse(
                                 const StudentInCourse& aStdnInCrs );
friend void StudentInCourseCollection::delete1StudentInCourse(
```

```
                                        int aIdNum, string code );
public:
   StudentCollection();
   ~StudentCollection()  {  delete[] scArr;  delete[] scIndex;  };
// getters
   size_t getNOfStudents() const {  return scNOfStudents;  }
   const Student* getArr() const {  return scArr;  }
   const StudentInCourseCollection* getPAllEnrollments() const
   {  return scPAllEnrollments;  }
// setters
   void setPAllEnrollments( StudentInCourseCollection*
                                                  pEnrollments )
   {  scPAllEnrollments = pEnrollments;  }
// 1 Student
   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) >= 0 );  };
   const Student& get1Student( int aIdNum ) const;
   void add1Student( const Student& aStudent );
   void delete1Student( int aIdNum );
// other
   void save( ofstream& bout, IndexEntry* index );
   void load( ifstream& bin );
   void swapArr( StudentCollection& rhs );
   void display( ostream& tout ) const;
private:
   enum { scIncr = 30 };
   Student*                   scArr;
   size_t                     scNOfStudents;
   size_t                     scReserved;
   StudentInCourseCollection* scPAllEnrollments;

   StudentCollection( const StudentCollection& rhs ) { };
   StudentCollection& operator=( const StudentCollection& rhs ) { };
   void erase1Student( int ndx );
   void insert1Student( const Student& aStudent );
   void add1Course( int aIdNum, const Course& aCourse );
   void delete1Course( int aIdNum, const Course& aCourse );
   int findNdx( int aIdNum ) const;
}; // StudentCollection
```

Η αντίστοιχη κλάση εξαιρέσεων:

```
struct StudentCollectionXptn
{  // version 1
   enum { allocFailed, notFound, noEnroll, enrollRef,
          fileNotOpen, cannotWrite, cannotRead };
   char funcName[100];
   int  errorCode;
   char errStrVal[100];
   char errIntVal;
   StudentCollectionXptn( const char* mn, int ec,
                          const char* sv="", int iv=0  )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  };
   StudentCollectionXptn( const char* mn, int ec, int iv )
      :  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  };
}; // StudentCollectionXptn
```

Prj04.11 Το 1ο Πρόγραμμα – Δημιουργία

Κρατούμε κατά βάση το σχέδιο που χρησιμοποιήσαμε στην πρώτη εκδοχή του προγράμματος (Prj03.2)· η υλοποίηση όμως θα είναι διαφορετική και μάλλον πιο απλή αφού αρκετή πολυπλοκότητα «κρύφτηκε» μέσα στις κλάσεις και στις μεθόδους τους.

Ας ξεκινήσουμε με τη

```
void loadCourses( string flNm, CourseCollection& allCourses )
{
// Άνοιξε το αρχείο
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "loadCourses", ProgXptn::cannotOpen,
                      flNm.c_str() );
// Τώρα διάβαζε
   Course oneCourse;
   loadSylCourse( bin, oneCourse );
   while ( !bin.eof() )
   {
      oneCourse.clearStudents();
      allCourses.add1Course( oneCourse );
      loadSylCourse( bin, oneCourse );
   } // while
   bin.close();
} // loadCourses
```

(όπου *loadSylCourse()* είναι –χωρίς αλλαγές– αυτή που είδαμε στο πρόγραμμα του Project 3.)

Εδώ βλέπεις τι εννοούμε: όλη η διαχείριση δυναμικής μνήμης κρύφτηκε στις μεθόδους της *allCourses*.

Prj04.11.1 Αρχείο Φοιτητών και Δηλώσεων Μαθημάτων

Ας δούμε τώρα πώς θα διαβάσουμε το αρχείο `enrllmnt.txt`:

- Μπορούμε να διαβάσουμε *όλα* τα στοιχεία (μαζί με τους κωδικούς μαθημάτων που δήλωσε) που αφορούν έναν φοιτητή και να τα εισαγάγουμε στο Μητρώο Φοιτητών αν δεν υπάρχει ήδη μέσα εγγραφή για τον συγκεκριμένο φοιτητή.
- Μπορούμε να διαβάσουμε τα στοιχεία του φοιτητή –*αλλά όχι τους κωδικούς των μαθημάτων*– και να τα εισαγάγουμε στο Μητρώο Φοιτητών. Στη συνέχεια διαβάζουμε και εισάγουμε –στο αρχείο δηλώσεων– έναν προς έναν τους κωδικούς μαθημάτων μαζί με τον αριθμό μητρώου του φοιτητή.

  Ας δούμε τι θα γίνεται με τα λάθη στην *πρώτη περίπτωση*:
- Αν στο Μητρώο Φοιτητών βρεθεί ο αριθμός μητρώου δεν γίνεται η εισαγωγή και προχωρούμε στον επόμενο φοιτητή.
- Αν στο αντικείμενο του φοιτητή βρεθεί κωδικός μαθήματος που δεν υπάρχει στον πίνακα μαθημάτων θα πάρουμε από την *add1StudentInCourse()* –μέσω της *add1Student()*– εξαίρεση. Στην περίπτωση αυτή θα πρέπει να διαγράψουμε τον κωδικό από το αντικείμενο του φοιτητή και να δοκιμάσουμε πάλι την εισαγωγή. Αν πάρουμε και πάλι *unknownCrs* διαγράφουμε τον κωδικό και ξαναδοκιμάζουμε. Αυτή η διαδικασία επαναλαμβάνεται μέχρις ότου γίνει η εισαγωγή. Για να καταλάβεις καλύτερα αυτό που γίνεται, ας πούμε ότι έχουμε δηλώσεις 6 (0..5) μαθημάτων και έχουμε λάθος στους κωδικούς 2, 3 και 5.
  - Την πρώτη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1 και 2. Διαγράφουμε τον κωδικό στη θέση 2 και εκεί έρχεται αυτός που ήταν στη θέση 5. Ο πίνακας μένει με 5 (0..4) στοιχεία.
  - Τη δεύτερη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1 και 2. Διαγράφουμε τον κωδικό στη θέση 2 και εκεί έρχεται αυτός που ήταν στη θέση 4. Ο πίνακας μένει με 4 (0..3) στοιχεία.
  - Την τρίτη φορά: γίνονται αναζητήσεις για τους κωδικούς 0, 1, 2 και 3. Διαγράφουμε τον κωδικό στη θέση 3.
  - Την τέταρη φορά: γίνονται (επιτυχείς) αναζητήσεις για τους κωδικούς 0, 1 και 2.

Μπορούμε να αποφύγουμε τα παραπάνω αν ελέγχουμε τους κωδικούς των μαθημάτων μόλις τους διαβάσουμε και εισάγουμε στον πίνακα του αντικειμένου μόνον τους σωστούς. Έτσι το αντικείμενο εισάγεται στο μητρώο φοιτητών με την πρώτη. Στο παραπάνω παράδειγμα εισάγουμε μόνον τους κωδικούς 0, 1 και 3. Πάντως θα πρέπει να πάρεις υπόψη σου ότι αυτοί οι έλεγχοι θα (ξανα)γίνουν όταν καλέσουμε την *add1Student()*.

Ερχόμαστε τώρα στη διαχείριση των σφαλμάτων στη *δεύτερη περίπτωση*:

- Αν δεν βρεθεί ο αριθμός μητώου η εισαγωγή στο Μητρώο Φοιτητών γίνεται αφού δεν υπάρχουν άλλα προβλήματα. Αν βρεθεί ο αριθμός μητρώου δεν γίνεται η εισαγωγή αλλά έχουμε μια διαφορά από την πρώτη περίπτωση: πριν προχωρήσουμε στον επόμενο φοιτητή θα πρέπει να διαβάσουμε και να αγνοήσουμε τους κωδικούς των μαθημάτων.
- Αν εισαχθεί ο φοιτητής είσαγουμε και τις δηλώσεις μαθημάτων, καλώντας την *add1StudentInCourse()*, για το καθένα. Αν για κάποιο ακυρωθεί η εισαγωγή πιάνουμε την εξαίρεση και γράφουμε στο *log*. Εδώ οι έλεγχοι που θα γίνουν είναι μόνον αυτοί της *add1StudentInCourse()* και των συναρτήσεων που αυτή καλεί.

Όπως καταλαβαίνεις ο δεύτερος τρόπος είναι προτιμότερος· αυτόν θα υλοποιήσουμε.

Πριν από αυτό όμως θα πρέπει να δούμε τις εξαιρέσεις που μας ενδιαφέρουν, δηλαδή αυτές για τις οποίες θα γράψουμε καταχώριση στο *log*:

- Στην *add1StudentInCourse()* το πρώτο πράγμα που κάνουμε είναι να αναζητήσουμε την εγγραφή σε ένα αντικείμενο κλάσης *StudentInCourseCollection*. Για την αναζήτηση (*findNdx()*) θα πρέπει να δημιουργηθεί –από τον αριθμό μητρώου του φοιτητή και τον κωδικό του μαθήματος– ένα αντικείμενο κλάσης *StudentInCourse*. Αν η δομή του κωδικού μαθήματος δεν συμμορφώνεται με τους κανόνες (το μήκος να είναι 7)[10] *ο δημιουργός της StudentInCourse* θα ρίξει εξαίρεση *StudentInCourseXptn* με κωδικό *keyLen*.
- Αν δεν υπάρχει πρόβλημα με τη δομή του κωδικού μαθήματος γίνεται αναζήτηση στον πίνακα μαθημάτων· αν δεν βρεθεί μάθημα με τέτοιον κωδικό ρίχνεται εξαίρεση *StudentInCourseCollectionXptn* με κωδικό *unknownCrs* από την *add1StudentInCourse*.

Θα διαβάσουμε το αρχείο με τα στοιχεία των φοιτητών με την:

```
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log )
{
   ifstream tin( flNm.c_str() ); //"enrllmnt.txt" );
   if ( tin.fail() )
      throw ProgXptn( "readStudentData", ProgXptn::cannotOpen, flNm.c_str() );
   do {
      readAStudent( tin, allStudents, allEnrollments, allCourses, log );
   } while( !tin.eof() );
   tin.close();
} // readStudentData
```

που καλεί την παρακάτω συνάρτηση για να διαβάσει τα στοιχεία του κάθε φοιτητή και να ενημερώσει καταλλήλως τους πίνακες:

```
void readAStudent( istream& tin,
                   StudentCollection& allStudents,
                   StudentInCourseCollection& allEnrollments,
                   CourseCollection& allCourses,
                   ofstream& log )
{
   Student      aStudent;
   unsigned int sIdNum;

// Διάβασε τα στοιχεία ενός φοιτητή
```

[10] ή κάτι πιο πολύπλοκο, σαν αυτά που έχουμε στην άσκ. Prj03-1.

```
   aStudent.readPartFromText( tin );
   if ( !tin.eof() )
   {
      sIdNum = aStudent.getIdNum();
      if ( allStudents.find1Student(aStudent.getIdNum()) )
      { // υπάρχει στον πίνακα φοιτητών
      // Αγνόησε τη δήλωση
         ignoreStudentData( tin );
         log << " multiple entry for student with id num "
             << sIdNum << endl;
      }
      else
      {
      // Βάλε τον φοιτητή στον πίνακα φοιτητών
         allStudents.add1Student( aStudent );
         string str1;
         getline( tin, str1, '\n' );
         if ( !tin.eof() )
         {
            int noc( atoi(str1.c_str()) );
            for ( int k(0); k < noc && !tin.eof(); ++k )
            {
               getline( tin, str1, '\n' );
               try
               {
                  StudentInCourse aStdnInCrs( sIdNum, str1 );
                  allEnrollments.add1StudentInCourse( aStdnInCrs);
               }
               catch( StudentInCourseXptn& x )
               {
                  if ( x.errorCode == StudentInCourseXptn::keyLen )
                     log << " student with id num "
                         << aStudent.getIdNum()
                         << " asking course " << str1 << endl;
                  else
                     throw;
               } // catch( StudentInCourseXptn...
               catch( StudentInCourseCollectionXptn& x )
               {
                  if ( x.errorCode ==
                                 StudentInCourseCollectionXptn::unknownCrs )
                     log << " student with id num " << aStudent.getIdNum()
                         << " asking course " << str1 << endl;
                  else
                     throw;
               } // catch( StudentInCourseCollectionXptn...
            } // for
         } // if ( !tin.eof() )
         if ( !tin.eof() ) getline( tin, str1, '\n' );//blank line
      } // if ( allStudents.find1Student(aStudent.getIdNum()) )
   } // if ( !tin.eof() )
} // readAStudent
```

Η κάθε μια `catch` «μεταφράζει» σε C++ αυτά που είπαμε παραπάνω για τις εξαιρέσεις. Πρόσεξε ότι:

- Οι εξαιρέσεις άλλων τύπων –εκτός από τους δύο που μας ενδιαφέρουν– περνούν «ανενόχλητες».
- Ακόμη περνούν –αλλά με τη "`throw;`"– οι εξαιρέσεις που πιάνουμε αλλά έχουν κωδικό άλλον από αυτόν που μας ενδιαφέρει.

Prj04.11.2 Έλεγχος Δηλώσεων

Η πιο «φυσική» θέση της συνάρτησης για τον έλεγχο των δηλώσεων μαθημάτων (εβδομαδιαίου φόρτου) είναι ως μια μέθοδος της *StudentCollection*:

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      if ( scArr[k].getWH() > maxWH )
         log << "student with id num " << scArr[k].getIdNum() << ": "
             << scArr[k].getWH() << " hours/week" << endl;
   } // for
} // StudentCollection::checkWH
```

Αυτή καλείται από τη **main**, μετά την ανάγνωση των δεδομένων όλων των φοιτητών:

```
      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );
```

Prj04.11.3 Φύλαξη

Η φύλαξη των συλλογών (και του ευρετηρίου) γίνεται με κλήση της

```
void saveCollections( CourseCollection& allCourses,
                      StudentCollection& allStudents,
                      SIndexEntry* index,
                      StudentInCourseCollection& allEnrollments )
{
   ofstream bout( "Courses.dta", ios_base::binary );
   allCourses.save( bout );
   bout.close();
   bout.open( "students.dta", ios_base::binary );
   allStudents.save( bout, index );
   bout.close();
   bout.open( "students.ndx", ios_base::binary );
   for ( int k(0); k < allStudents.getNOfStudents(); ++k )
      bout.write( reinterpret_cast<const char*>(&index[k]),
                  sizeof(SIndexEntry) );
   bout.close();
   bout.open( "enrllmnt.dta", ios_base::binary );
   allEnrollments.save( bout );
   bout.close();
} // saveCollections
```

Αφού το ευρετήριο είναι πίνακας μπορεί να φυλαχθεί με πιο απλό τρόπο:

```
   bout.write( reinterpret_cast<const char*>(index),
               allStudents.getNOfStudents()*sizeof(IndexEntry) );
```

Prj04.11.4 ...Και το Πρόγραμμα

Το πρόγραμμα θα είναι ως εξής:

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

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "StudentInCourse.cpp"
#include "StudentInCourseCollection.h"
#include "CourseCollection.h"
#include "SIndexEntry.h"
```

```
#include "StudentCollection.h"
#include "CourseCollection.cpp"
#include "StudentCollection.cpp"
#include "StudentInCourseCollection.cpp"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadCourses( string flNm, CourseCollection& allCourses );
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log );
void saveCollections( CourseCollection& allCourses,
                      StudentCollection& allStudents,
                      IndexEntry* index,
                      StudentInCourseCollection& allEnrollments );

int main()
{
   try
   {
      CourseCollection allCourses;
      StudentCollection allStudents;
      StudentInCourseCollection allEnrollments;

      allEnrollments.setPAllStudents( &allStudents );
      allEnrollments.setPAllCourses( &allCourses );

      allCourses.setPAllEnrollments( &allEnrollments );
      allStudents.setPAllEnrollments( &allEnrollments );

      loadCourses( "Courses.dta", allCourses );

      ofstream log( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt");

      readStudentData( "enrllmnt.txt", allStudents, allEnrollments,
                       allCourses, log );

      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );

      log.close();

      SIndexEntry* index;
      try
      {  index = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }

      saveCollections( allCourses, allStudents, index, allEnrollments );

      delete[] index;
   } // try
```

```
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( CourseXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )
   {
      switch ( x.errorCode )
      {
         case StudentXptn::allocFailed:
            cout << "Student " << x.objKey << ": no dynamic memory in "
                 << x.funcName << endl;
            break;
         case StudentXptn::negIdNum:
            cout << "Student " << x.objKey << ": illegal id num ("
                 << x.errIntVal << ") in " << x.funcName << endl;
            break;
         case StudentXptn::incomplete:
            cout << "incomplete data for Student " << x.objKey << " in "
                 << x.funcName << endl;
            break;
         case StudentXptn::fileNotOpen:
            cout << "file " << x.errStrVal << " not open in " << x.funcName
                 << endl;
            break;
         case StudentXptn::cannotWrite:
            cout << "cannot write to file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         case StudentXptn::cannotRead:
            cout << "cannot read from file " << x.errStrVal << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected StudentXptn for Student " << x.objKey
                 << " from " << x.funcName << endl;
      } // switch
   } // catch( StudentXptn
   catch( StudentInCourseXptn& x )  { /* . . . */ }
   catch( CourseCollectionXptn& x )  { /* . . . */ }
   catch( StudentCollectionXptn& x )  { /* . . . */ }
   catch( StudentInCourseCollectionXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Να παρατηρήσουμε τα εξής:

- Αν αλλάξεις τη σειρά των οδηγιών "`include`" μπορεί να αντιμετωπίσεις προβλήματα.
- Η "`delete[] index`" είναι απαραίτητη; Σε κάθε περίπτωση, αφού εκεί τελειώνει το πρόγραμμα, η δυναμική μνήμη θα ελευθερωθεί! Σωστό! Τη βάλαμε μόνο και μόνο για να δεις ότι ενώ για την ανακύκλωση των δυναμικών πινάκων που είναι μέσα σε αντικείμενα θα δράσουν οι καταστροφείς για τον index θα πρέπει να ζητήσουμε ρητώς την ανακύκλωση.
- Όπως καταλαβαίνεις, δεν υπάρχει λόγος να παραθέσουμε το μακροσκελέστατο τμήμα διαχείρισης εξαιρέσεων –δηλαδή όλες τις `catch`– αφού οι μόνες που έχουν πραγματικό ενδιαφέρον είναι αυτές της *readAStudent* και όχι αυτές της **main**.
- Τέλος, πρόσεξε πόσο απλή είναι η **main** αφού όλη η πολυπλοκότητα έχει κρυφτεί μέσα στις μεθόδους.

Prj04.12 Το 2ο Πρόγραμμα – Εκμετάλλευση

Το δεύτερο πρόγραμμα είναι πολύ απλό:

```
#include <fstream>
```

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

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "SIndexEntry.h"

struct ProgXptn
// ΟΠΩΣ ΣΤΟ 1ο ΠΡΟΓΡΑΜΜΑ

void loadIndex( string flNm, PIndexEntry& index, unsigned int& ndxSz );
void retrieve( string flNm, PIndexEntry index, int ndxSz );

int main()
{
   IndexEntry*  index;
   unsigned int ndxSz;
   try
   {
      loadIndex( "students.ndx", index, ndxSz );
      retrieve( "students.dta", index, ndxSz );
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
}
```

Στο **SIndexEntry.h** έχουμε τα εξής:

```
struct SIndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
   explicit IndexEntry( int aIdNum=0, int aLoc=0 )
   {  sIdNum = aIdNum;  loc = aLoc;  }
}; // SIndexEntry

typedef SIndexEntry* PIndexEntry;

bool operator!=( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return ( lhs.sIdNum != rhs.sIdNum );  }

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

Επιφορτώνουμε τον "**!=**" (και τον "**==**") και γράφουμε τον ερήμην δημιουργό με αρχικές τιμές διότι θα μας χρειαστούν για να χρησιμοποιήσουμε τη *linSearch()*. Δύο αντικείμενα *SIndexEntry* θεωρούνται ίσα αν και μόνον αν έχουν ίδια τιμή στο *sIdNum*. Φυσικά, θα παρατηρήσεις ότι ο δημιουργός χρειάζεται κάποιους ελέγχους. Σωστό! Γράψε τους (μαζί και μια κλάση εξαιρέσεων)!

Ας δούμε τώρα τις δύο συναρτήσεις. Η *loadIndex()* βασίζεται σε αυτά που μάθαμε στην §15.12.1:

```
void loadIndex( string flNm, PIndexEntry& index, unsigned int& ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "loadIndex", ProgXptn::cannotOpen, flNm.c_str() );

   bin.seekg( 0, ios_base::end );
   unsigned int flSize( bin.tellg() );
```

```
   ndxSz = flSize/sizeof(IndexEntry);

   try {  index = new IndexEntry[ndxSz+1];  }
   catch( bad_alloc )
   {  throw ProgXptn( "loadIndex", ProgXptn::allocFailed );  }

   bin.seekg( 0 );
   for ( int k(0); k < ndxSz; ++k )
      bin.read( reinterpret_cast<char*>(&index[k]), sizeof(IndexEntry) );
   bin.close();
} // loadIndex
```

Με τις:

```
   bin.seekg( 0, ios_base::end );
   unsigned int flSize( bin.tellg() );
   ndxSz = flSize/sizeof(IndexEntry);
```

βρίσκουμε το μέγεθος του αρχείου και υπολογίζουμε το πλήθος των στοιχείων που περιέχονται σε αυτό.

Στη συνέχεια παίρνουμε την απαιτούμενη δυναμική μνήμη. Το "**+1**" μας δίνει μια θέση στον πίνακα για τον φρουρό που χρησιμοποιεί η *linSearch()*.

Τέλος, με τη **for** φορτώνουμε το περιεχόμενο του αρχείου στον πίνακα.

Η ανάκτηση και η επίδειξη των στοιχείων των φοιτητών γίνεται με τη:

```
void retrieve( string flNm, PIndexEntry index, int ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "retrieve", ProgXptn::cannotOpen, flNm.c_str() );

   string line;
   cout << "Student Id Number: ";  getline( cin, line, '\n' );
   while ( line != "ΤΕΛΟΣ" )
   {
      int idNum( atoi(line.c_str()) );
      int ndx( linSearch(index, ndxSz, 0, ndxSz-1, IndexEntry(idNum)) );
      if ( ndx < 0 )
         cout << "unknown Student Id Number" << endl;
      else
      {
         Student oneStudent;
         bin.seekg( index[ndx].loc );
         oneStudent.load( bin );
         oneStudent.display( cout );
      }
      cout << "Student Id Number: ";  getline( cin, line, '\n' );
   } // while
   bin.close();
} // retrieve
```

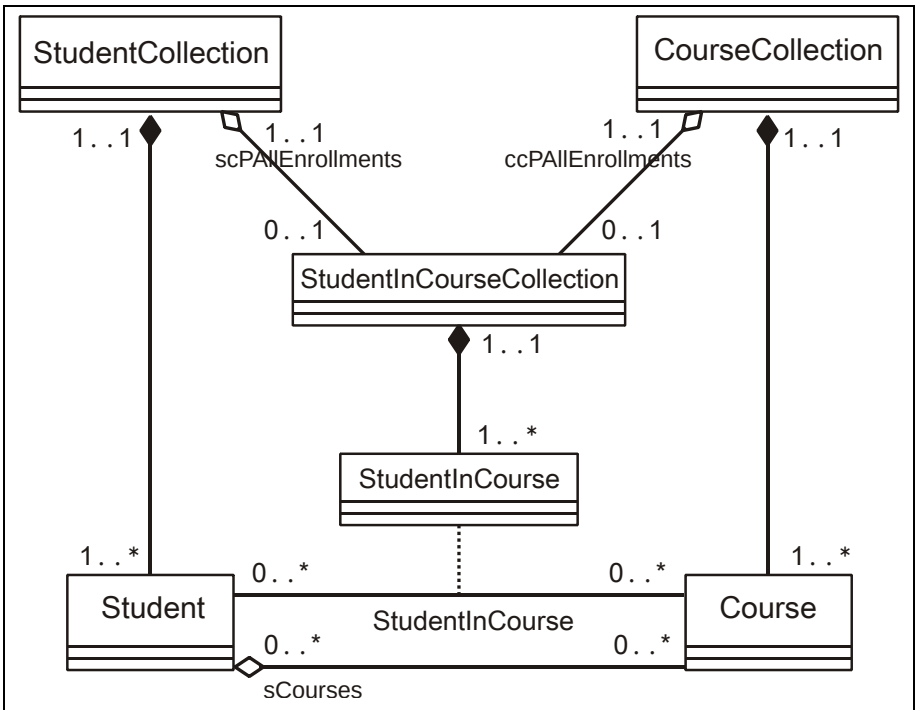
Η *retrieve()* τροφοδοτείται με τον index και το όνομα του αρχείου –με τα στοιχεία των φοιτητών– το οποίο και ανοίγει.

Στη συνέχεια ζητάει από τον χρήστη τον αριθμό μητρώου ενός φοιτητή και ψάχνει στον index να βρει στοιχείο με αυτόν. Αν το βρει –στη θέση *ndx*– φορτώνει τα στοιχεία του φοιτητή με τις:

```
         Student oneStudent;
         bin.seekg( index[ndx].loc );
         oneStudent.load( bin );
```

Στην §15.14.3 είχαμε υποσχεθεί ότι «*Αργότερα θα δούμε μια πολύ απλή μορφή ευρετηρίου.*» Εδώ πραγματοποιήσαμε αυτό που υποσχεθήκαμε. Αλλά... το παράδειγμά μας έχει πολύ περιορισμένη χρησιμότητα: Μόνο ανάκτηση πληροφοριών.

Δεν μπορούμε να κάνουμε και ενημέρωση; Μπορούμε, αρκεί να περιοριστούμε στα γενικά στοιχεία του φοιτητή και να μην αλλάξουμε το πλήθος μαθημάτων. Φυσικά δεν μπο-



Σχ. Prj04-1 Διάγραμμα κλασεων του παραδείγματος.

ρούμε να διαγράφουμε ή να προσθέτουμε νέους φοιτητές. Με τέτοιες ενημερώσεις χάνουμε τη δυνατότητα της σειριακής διαχείρισης του αρχείου. Για να τα έχουμε όλα χρειάζονται πέρα από το ευρετήριο και ειδικοί αλγόριθμοι που δεν έχουν σχέση με τους στόχους αυτού του βιβλίου.

Prj04.13 Για το Παράδειγμά μας

Με αυτό το παράδειγμα είχαμε την ευκαιρία να δούμε πώς χρησιμοποιούμε αυτά που μάθαμε μέχρι τώρα για κλάσεις και αντικείμενα. Στο Σχ. Prj04-1 βλέπεις και το σχετικό διάγραμμα κλάσεων.

Πόσο «πραγματικό» είναι το παράδειγμα; Όχι πολύ. Και αυτό δεν έχει σχέση μόνο με τα στοιχεία που θα πρέπει να κρατάει ένα αρχείο φοιτητών και λείπουν από το δικό μας. Ας δούμε τα άλλα προβλήματά του.

- Μια «πραγματική» εφαρμογή αυτού του είδους στήνεται με ένα *Σύστημα Διαχείρισης Βάσεων Στοιχείων* (Data Base Management Systems, DBMS) και όχι με απλά αρχεία. Πολλά από τα προβλήματα που αντιμετωπίσαμε[11] (και μερικά από αυτά που θα δούμε στη συνέχεια) αντιμετωπίζονται από το ΣΔΒΔ και η δουλειά του προγραμματιστή διευκολύνεται. Φυσικά, το ΣΔΒΔ κρατάει τους πίνακας στη ΒΔ, στον δίσκο, και όχι στη μνήμη.
- Το «σωστό» σχέδιο ήταν αυτό που είχαμε στο Project 3 και όχι το τωρινό.[12] Η αλήθεια είναι ότι στις «πραγματικές» εφαρμογές γίνονται τέτοιες «αποκανονικοποιήσεις» για να αυξήσουμε την ταχύτητα εκτέλεσης. Αλλά αυτές εισάγουν πλεονασμό και είδες πώς πληρώνεται ο πλεονασμός: οι μεγαλύτερες περιπλοκές σε αυτά που γράψαμε έχουν σχέση με τη συνεπή ενημέρωση του πίνακα δηλώσεων μαθημάτων και του πίνακα μαθημάτων του κάθε φοιτητή.

---

[11] Για παράδειγμα οι έλεγχοι ακεραιότητας οντότητας και αναφοράς.

[12] Στη γλώσσα του Σχεσιακού μοντέλου ΒΔ λέμε ότι ο πίνακας μαθημάτων μέσα στο αντικείμενο κλάσης *Student* παραβιάζει την *1η Κανονική Μορφή*.

- Θα μπορούσαμε να έχουμε τον πλεονασμό μόνο στην κύρια μνήμη αλλά όχι στα αρχεία: Φυλάγουμε στο αρχείο φοιτητών για κάθε φοιτητή μόνον επώνυμο, όνομα και αριθμό μητρώου. Κατα τη φόρτωση των στοιχείων, τα υπόλοιπα συμπληρώνονται όταν φορτώνουμε το αρχείο δηλώσεων μαθημάτων. Έτσι δεν έχουμε εγγραφές μεταβλητού μήκους και δεν χρειαζόμαστε ευρετήριο (τουλάχιστον για τον λόγο αυτόν).
- Και κάτι ακόμη: Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
```

- Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)

- Σε ένα τέτοιο σύστημα προγραμμάτων και δεδομένων, σε πραγματικές συνθήκες, θα πρέπει να έχουν ταυτόχρονη πρόσβαση πολλοί χρήστες –για παράδειγμα υπάλληλοι της γραμματείας του τμήματος και πιθανότατα οι ίδιοι οι ενδιαφερόμενοι φοιτητές– χωρίς όμως να καταστρέφουν ο ένας τη δουλειά του άλλου. Οι κίνδυνοι προέρχονται από τον τρόπο λειτουργίας των πολυχρηστικών ΛΣ που δίνουν εκ περιτροπής χρόνο εξυπηρέτησης σε όλους τους χρήστες αλλά με πολύ μεγάλη –για τον άνθρωπο– ταχύτητα ώστε να μην ενοχλείται ο χρήστης. Σκέψου λοιπόν την εξής περίπτωση:
  - Υπάλληλος της γραμματείας ζητάει φύλαξη των πινάκων (με τη συνάρτηση *saveCollections()*) του πρώτου προγράμματος· φυλάγεται ο πίνακας μαθημάτων, φυλάγεται το μητρώο των φοιτητών και…
  - Στο σημείο εκείνο το ΛΣ διακόπτει την εκτέλεση του προγράμματος και εξυπηρετεί την απαίτηση ενός φοιτητή που αλλάζει τις δηλώσεις μαθημάτων.
  - Όταν θα επιστρέψει για να συνεχίσει τη φύλαξη θα φυλάξει τον πίνακα δηλώσεων που όμως δεν είναι συνεπής με αυτούς που είχαν φυλαχθεί πριν τη διακοπή.

  Ένα ΣΔΒΔ, με τη δυνατότητα **επεξεργασίας συναλλαγών** (transaction processing) που σου παρέχει, σου επιτρέπει να απαλλαγείς με σχετικώς απλό τρόπο από τέτοιες επι-

πλοκές. Αλλιώς θα πρέπει να καταφύγεις στη χρήση εργαλείων που δίνει το ΛΣ (σημαφόροι, κλειδώματα κ.ά.) που κάνουν το πρόγραμμά σου πολύ πιο πολύπλοκο.

project

5

Σύνολα Γραμμάτων

Περιεχόμενα:

Prj05.1 Το Πρόβλημα 825
Prj05.2 Παίρνοντας Ιδέες από την Pascal 826
Prj05.3 Η Κλάση και οι Μέθοδοι 827
- **Prj05.3.1 Πληθάριθμος: #*x*** 828
- **Prj05.3.2 Ένωση Συνόλων - Εισαγωγή Στοιχείου σε Σύνολο** 829
- **Prj05.3.3 Διαφορά Συνόλων - Διαγραφή Στοιχείου Συνόλου** 830
- **Prj05.3.4 Τομή Συνόλων: *x* ∩= *y*** 832

Prj05.4 Το Πρόγραμμα 832
- **Prj05.4.1 Και η *SetOfUCL* Μέχρι Τώρα** 835

Prj05.5 Εμπλουτίζοντας την Κλάση 836
Prj05.6 Σχόλια και Παρατηρήσεις 838

Prj05.1 Το Πρόβλημα

Το Project αυτό είναι ένα μεγάλο παράδειγμα επιφόρτωσης τελεστών.

Α. Θέλουμε να υλοποιήσουμε έναν τύπο συνόλων που στοιχεία τους θα είναι Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου. Για την υλοποίηση επιλέγουμε τη χρήση ψηφιοχάρτη (bitmap). Δηλαδή ένα σύνολο θα παριστάνεται με τα πρώτα 26 bits μιας τιμής τύπου `long int` *και:*

– το bit 0 έχει τιμή 1 αν και μόνον αν το `'A'` *ανήκει στο σύνολο,*

– το bit 1 έχει τιμή 1 αν και μόνον αν το `'B'` *ανήκει στο σύνολο,*

. . .

– το bit 25 έχει τιμή 1 αν και μόνον αν το `'Z'` *ανήκει στο σύνολο,*

Αν για παράδειγμα έχουμε μια τιμή:

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1
```

αυτή παριστάνει το σύνολο { `'A'`, `'C'`, `'D'`, `'F'`, `'G'`, `'Z'` }. *Προφανώς το* { } *παριστάνεται με την τιμή 0.*

Γράψε μια κλάση, ας την πούμε SetOfUCL,[1] *που θα υλοποιεί το σύνολό μας με τον παραπάνω τρόπο. Η κλάση θα περιέχει μεθόδους για:*

(x, y σύνολα, u άτομο)

– πληθάριθμο: **#***x,*

– εισαγωγή στοιχείου σε σύνολο: x **+=** *u και x* **+** *u,*

– διαγραφή στοιχείου από σύνολο: x **~=** *u και x* **~** *u,*

[1] *Set of* Upper Case Letters

– ένωση συνόλων: x ∪= y και x ∪ y,
– τομή συνόλων: x ∩= y και x ∩ y,
– διαφορά συνόλων: x \= y και x \ y,
Φυσικά θα πρέπει να γράψεις και όποιες άλλες μεθόδους ή καθολικές συναρτήσεις θεωρείς αναγκαίες.
Υπόδ.: Για ευκολία μπορείς να υποθέσεις ότι όλα τα Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου βρίσκονται σε συναπτές θέσεις του πίνακα χαρακτήρων κατά αλφαβητική σειρά: αν το `'A'` *βρίσκεται στη θέση p, τότε το* `'B'` *βρίσκεται στη θέση p+1, το* `'C'` *βρίσκεται στη θέση p+2 κ.ο.κ.*
Δίνεται αρχείο text, με όνομα στον δίσκο `mstrpc.txt`*, που περιέχει κείμενο στην αγγλική γλώσσα. Τα γράμματα είναι κεφαλαία και πεζά αλλά για μας είναι ίδια:* `'A'` *και* `'a'` *είναι το ίδιο πράγμα (μια καλή ιδέα είναι η εξής: όταν τα διαβάζουμε, πριν από οποιαδήποτε άλλη δουλειά, τα κάνουμε κεφαλαία). Το κείμενο είναι χωρισμένο σε παραγράφους. Κάθε παράγραφος (εκτός από την τελευταία) χωρίζεται από την επόμενή της με μια κενή γραμμή (δηλαδή: αν διαβάσουμε* `'\n''\n'` *αλλάζει παράγραφος).*
Θέλουμε ένα πρόγραμμα που θα δημιουργεί ένα άλλο αρχείο, μη μορφοποιημένο, με όνομα στο δίσκο `paragltr.dta`*, που για κάθε παράγραφο θα έχει τρεις τιμές κλάσης setOfUCL:*
– το σύνολο των γραμμάτων της παραγράφου (τα βλέπουμε όλα ως κεφαλαία),
– το σύνολο των γραμμάτων της παραγράφου που υπάρχουν και στην επόμενη παράγραφο,
– το σύνολο των γραμμάτων της παραγράφου που δεν υπάρχουν στην επόμενη παράγραφο.

Prj05.2 Παίρνοντας Ιδέες από την Pascal

Πριν προχωρήσουμε στην υλοποίηση της κλάσης, ας ρίξουμε μια ματιά στη διαχείριση συνόλων που επιτρέπει η Pascal που επιτρέπει δηλώσεις της μορφής:

```
x, y, z: set of T;
```

Μετά από αυτό οι *x*, *y*, *z* είναι σύνολα με στοιχεία τύπου *T* (τύπος βάσης). Οι πράξεις συνόλων γίνονται με τους τελεστές:

- "`+`" για την ένωση. Το "`x + y`" παριστάνει το "*x* ∪ *y*"
- "`*`" για την τομή. Το "`x * y`" παριστάνει το "*x* ∩ *y*"
- "`-`" για τη διαφορά. Το "`x - y`" παριστάνει το "*x* \ *y*"

Με τις "`x = y`" και "`x <> y`" συγκρίνονται τα *x*, *y* για ισότητα και ανισότητα αντιστοίχως.

Αν *u* τιμή τύπου *T*, το "`u in x`" είναι συνθήκη που παίρνει τιμή `true` αν η *u* *ανήκει* στο *x* και `false` αν δεν ανήκει.

Συνθήκη είναι και η "`x <= y`" που παίρνει τιμή `true` αν το *x* είναι *υποσύνολο* του *y* και `false` αν δεν είναι.[2] Η σύγκριση μπορεί να γίνει και με την "`y >= x`" (το *y* είναι *υπερσύνολο* του *x*).

Στην Extended Pascal παρέχεται και η συνάρτηση *card*: Με το "`card(x)`" παίρνουμε τον **πληθάριθμο** (cardinality) του *x* (`#`*x*).

Δεν είναι κακή ιδέα να επιφορτώσουμε τους αντίστοιχους τελεστές με το ίδιο νόημα για την κλάση που έχουμε να γράψουμε.

---

[2] *Μην* υποθέσεις ότι με το "`x < y`" ελέγχουμε για *γνήσιο υποσύνολο*. Αν τα *x*, *y* είναι σύνολα η Pascal δεν επιτρέπει να γράψεις τέτοια παράσταση.

Prj05.3 Η Κλάση και οι Μέθοδοι

Ξεκινούμε με την αναλλοίωτη της κλάσης η οποία είναι τετριμμένη: `true`. Σύμφωνα με τους κανόνες μας θα πρέπει να γράψουμε `struct` (όλα ανοικτά) και όχι `class`:

```
struct SetOfUCL
{
   unsigned long int bitmap;
// . . .
}; // SetOfUCL
```

Με το πρόβλημα αυτό ασχοληθήκαμε στο Παράδ. 3 της §19.5 όπου λέγαμε «*Αλλά αν επιλέξουμε* `struct` *έχουμε το εξής πρόβλημα: Δίνουμε τη δυνατότητα στο πρόγραμμα που τη χρησιμοποιεί να χειρίζεται την τιμή του (μοναδικού) μέλους ενώ η κλάση γράφεται για να του δώσει τη δυνατότητα να χειρίζεται σύνολα και τα μέλη τους. Πώς αποφεύγεται κάτι τέτοιο;*

- *Με το να την κάνουμε* `class`,
- *να βάλουμε το "*`long int bitmap`*" («μυστικό» της υλοποίησης) σε περιοχή* `private` *και*
- *να μην γράψουμε μεθόδους getBitmap ούτε setBitmap.*»:

```
class SetOfUCL
{
public:
// . . .
private:
   unsigned long int bitmap;
}; // SetOfUCL
```

Όπως λέει και το όνομά της, θα χειριζόμαστε τη *bitmap* ως ψηφιοπίνακα με τις συναρτήσεις (*bitValue()*, *setBit()*, *clearBit()* κλπ) που μάθαμε στην §17.6.

Η πιο απλή μορφή της κλάσης, όπως την υποδεικνύει η διατύπωση των απαιτήσεων:

```
class SetOfUCL
{
public:
   SetOfUCL() {  bitmap = 0UL;  };
   ~SetOfUCL() { };
private:
   long int bitmap;
}; // SetOfUCL
```

- Όπως βλέπεις, ορίσαμε και τον ερήμην δημιουργό που μας δίνει το κενό σύνολο. Αν δηλαδή δηλώσουμε:

```
   SetOfUCL x;
// x == { }
```

  το *x* είναι κενό.

- Ο «κενός» καταστροφέας μας θυμίζει ότι δεν χρειάζεται να ορίσουμε δημιουργό αντιγραφής, τελεστή εκχώρησης και καταστροφέα μια και αυτοί που θα ορίσει αυτομάτως ο μεταγλωττιστής κάνουν τη δουλειά μας.

Είναι χρήσιμο να έχουμε και έναν δημιουργό (με αρχική τιμή) για δημιουργία *μονοσυνόλων*. Τον δηλώνουμε:

```
   explicit SetOfUCL( char c );
```

και τον γράφουμε αντιγράφοντας από τη *setBit()*.

Σε ποιο δυαδικό ψηφίο θα βάλουμε την τιμή "1"; Αφού το `'A'` αντιστοιχεί στο δυαδικό ψηφίο 0, ένα τυχόν κεφαλαίο γράμμα *c* θα αντιστοιχεί στο δυαδικό ψηφίο:

```
static_cast<int>(c) - static_cast<int>('A')
```

Και αν ο *c* δεν είναι κεφαλαίο γράμμα τί κάνουμε; Τότε *πρέπει να ρίξουμε εξαίρεση* αφού δεν μπορούμε να ανταποκριθούμε:

```
SetOfUCL::SetOfUCL( char c )
{
   if ( !isupper( c ) )
      throw SetOfUCLXptn( "SetOfUCL", SetOfUCLXptn::nonUCL, c );
```

```
   bitmap = 1;
   bitmap <<= ( static_cast<int>(c) - static_cast<int>('A') );
} // SetOfUCL::SetOfUCL
```

Η

```
   SetOfUCL oneSet( 'S' );
```

δημιουργεί ένα (μονο)σύνολο με μοναδικό στοιχείο το `'S'`.

Παρατηρήσεις: ▶

1. Προφανώς θα μπορούσαμε να γράψουμε:

```
   bitmap <<= ( c - 'A' );
```

αλλά προτιμούμε τη γραφή με την τυποθεώρηση για να σου υπενθυμίζουμε τι γίνεται. Είτε με τη μια γραφή είτε με την άλλη, βασιζόμαστε στα εξής:

- *«Για ευκολία μπορείς να υποθέσεις ότι όλα τα Κεφαλαία Γράμματα του Λατινικού Αλφαβήτου βρίσκονται σε συναπτές θέσεις του πίνακα χαρακτήρων κατά αλφαβητική σειρά: αν το* `'A'` *βρίσκεται στη θέση p, τότε το* `'B'` *βρίσκεται στη θέση p+1, το* `'C'` *βρίσκεται στη θέση p+2 κ.ο.κ.»*
- *«Το bit 0 έχει τιμή 1 αν και μόνον αν το* `'A'` *ανήκει στο σύνολο.»*

2. Αν δεν θέλεις να αντιγράψεις τη *setBit()* μπορείς απλώς να την καλέσεις:

```
SetOfUCL::SetOfUCL( char c )
{
   if ( !isupper( c ) )
      throw SetOfUCLXptn( "SetOfUCL", SetOfUCLXptn::nonUCL, c );

   bitmap = 0;
   setBit( bitmap, static_cast<int>(c) - static_cast<int>('A') );
} // SetOfUCL::SetOfUCL
```

Φυσικά, στην περίπτωση αυτή θα πρέπει να φέρεις στο πρόγραμμά σου το περίγραμμα *setBit()*.

Το περίγραμμα της *setBit()* μπορεί να ρίξει και κάποια εξαίρεση· μήπως πρέπει να την πιάσουμε και να ρίξουμε μια *SetOfUCLXptn*; Όχι! Δεν υπάρχει περίπτωση να τη ρίξει!◀

Μεθόδους "*get*" και "*set*" θα γράψουμε; Για τη *bitmap* ούτε λόγος. Πάντως τον ρόλο "*get*" θα παίξει το κατηγόρημα «*ανήκειν*» (∈) και ρόλο "*set*" θα παίξουν οι μέθοδοι *εισαγωγής στοιχείου σε σύνολο: x* `+=` *u* και *διαγραφής στοιχείου από σύνολο: x* `~=` *u*.

Όπως συνηθίζουμε, για να μπορούμε να κάνουμε τις δοκιμές μας με την κλάση, την εξοπλίζουμε με μια μέθοδο:

```
void SetOfUCL::display( ostream& tout ) const
{
   unsigned long int x( 1 );

   for ( int k(0); k <= 25; ++k )
   {
      if ( (bitmap & x) != 0 )
         tout << static_cast<char>( k + static_cast<int>('A') );
      x <<= 1;
   } // for
   tout << endl;
} // SetOfUCL::display
```

Prj05.3.1 Πληθάριθμος: #*x*

Η πρώτη μέθοδος που μας ζητείται είναι αυτή που δίνει τον *πληθάριθμο* ενός συνόλου *x* (#*x*). Ακολουθώντας την Extended Pascal την ονομάζουμε *card*:

x`.`*card*`()` `==` `#`*x*

Αφού για κάθε στοιχείο που περιέχεται στο σύνολο θα βάζουμε "1" στο αντίστοιχο δυαδικό ψηφίο του *bitmap*, αρκεί να μετρήσουμε πόσα "1" υπάρχουν στο *bitmap*. Επομένως, μπορούμε να χρησιμοποιήσουμε το περίγραμμα συνάρτησης *count1*:

```
unsigned int SetOfUCL::card() const
{  return count1( bitmap );  } // SetOfUCL::card
```

ή να το αντιγράψουμε με την κατάλληλη προσαρμογή:

```
unsigned int SetOfUCL::card() const
{
   unsigned long int x( 1 );
   int fv( 0 ), lastb( 8*sizeof(unsigned long int)-1 );

   for ( int k(0); k <= lastb; ++k )
   {
      if ( (bitmap & x) != 0 )  ++fv;
      x <<= 1;
   } // for
   return fv;
} // SetOfUCL::card
```

Prj05.3.2 Ένωση Συνόλων – Εισαγωγή Στοιχείου σε Σύνολο

Τι σχέση έχει η ένωση συνόλων με την εισαγωγή στοιχείου σε σύνολο; Η εισαγωγή είναι ειδική περίπτωση της ένωσης. Πράγματι, οι προδιαγραφές για την εισαγωγή είναι οι εξής:

`true` { εισαγωγή του u στο x } $u \in x$

Αλλά τις ίδιες προδιαγραφές έχει και η "$x = x \cup \{ u \}$" ($x \cup= \{ u \}$):

`true` { $x = x \cup \{ u \}$ } $u \in x$

Αν πάρουμε υπόψη μας ακόμη ότι από την "`operator@=`" μπορούμε να πάρουμε την υλοποίηση της "`operator@`" (επιφόρτωση του "`@`") καταλαβαίνεις ότι το πρώτο που έχουμε να κάνουμε είναι η υλοπόιηση του "$\cup$=". Και –αφού είπαμε ότι θα ακολουθήσουμε τους συμβολισμούς της Pascal– θα επιφορτώσουμε τον "`operator+=`". Όπως είπαμε στην §22.6 η επιφόρτωση θα πρέπει να γίνει με μέθοδο της κλάσης:

```
SetOfUCL& SetOfUCL::operator+=( const SetOfUCL& rhs )
{
   bitmap = bitmap | rhs.bitmap;
   return *this;
} // SetOfUCL::operator+=
```

Αν θέλεις, μπορείς να υλοποιήσεις και μια εκδοχή με όνομα αλλά πρόσεχε: *το "`union`" δεν επιτρέπεται*! Ας πούμε λοιπόν:

```
SetOfUCL& SetOfUCL::setUnion( const SetOfUCL& rhs )
{
   return ( *this += rhs );
} // SetOfUCL::setUnion
```

Πώς λύθηκε το πρόβλημα της εισαγωγής; Αν το x είναι αντικείμενο κλάσης *SetOfUCL* τότε με την

```
   x += SetOfUCL( 'S' );
```

υλοποιείς την $x = x \cup \{$ `'S'` $\}$.

Πώς λύθηκε το πρόβλημα της καθολικής συνάρτησης; Όπως μάθαμε στην §22.7.1:

```
SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv += rhs;
   return fv;
} // SetOfUCL operator+
```

Πρόσεξε ότι αυτή η συνάρτηση *δεν χρειάζεται να δηλωθεί ως* **friend**.

Η διατύπωση του προβλήματος όμως απαιτεί να κάνουμε εισαγωγή γράφοντας "`x += 'S'`". Μπορούμε να έχουμε αυτήν τη δυνατότητα; Ναι, με δύο τρόπους:

- Να βγάλουμε το "`explicit`" από τον δημιουργό μονοσυνόλου.
- Να κάνουμε δεύτερη επιφόρτωση του "`+=`".

Αν επιλέξουμε την πρώτη λύση θα είναι δυνατές (μετά την επιφόρτωση του "`==`") σύγκρισεις σαν την "`x == 'S'`" που είναι απαράδεκτη. Έτσι, θα πάμε στη δεύτερη λύση:

```
SetOfUCL& SetOfUCL::operator+=( char c )
{
   if ( !isupper(c) )
      throw SetOfUCLXptn( "operator+=", SetOfUCLXptn::nonUCL, c );

   *this += SetOfUCL( c );
   return *this;
} // SetOfUCL::operator+=
```

Και εδώ μπορούμε να δώσουμε και μια εκδοχή με όνομα:

```
SetOfUCL& SetOfUCL::insert( char c )
{
   return ( *this += c );
} // SetOfUCL::insert
```

Από τη μέθοδο εισαγωγής χαρακτήρα μπορείς να πάρεις –με την πάγια τεχνική– και καθολική συνάρτηση εισαγωγής χαρακτήρα:

```
SetOfUCL operator+( const SetOfUCL& lhs, char rhs )
```

Στα παραπάνω, πρόσεξε το εξής ουσιώδες: *Το νόημα του τελεστή "+" δίνεται μόνο μια φορά, στην επιφόρτωση του τελεστή "+=".* Όλοι οι άλλοι ορισμοί ανάγονται αμέσως ή εμμέσως στον "`+=`". Θα πει κανείς: Αν τα ορίσουμε όλα εξ αρχής, γράφοντας πράξεις με ψηφιοχάρτες, θα κάναμε τις συναρτήσεις μας πιο γρήγορες· και μόνο το ότι θα αποφεύγαμε τοσες κλήσεις συναρτήσεων θα ήταν σημαντικό κέρδος. Ναι, αλλά θα είχαμε περισσότερες πιθανότητες για λάθη και η νοηματική συνέπεια όλων αυτών δεν θα ήταν (σχεδόν) αυταπόδεικτη, όπως είναι τώρα.

Παράδειγμα ✍

Οι εντολές

```
   SetOfUCL x( 'A' ), y( x ), z;

   x += 'B';  x.insert( 'C' );
   x.display( cout );
   y += 'C';  y.insert( 'D' );  y += 'E';
   y.display( cout );
   z = x + y;
   z.display( cout );
   x.setUnion( y );
   x.display( cout );
   cout << x.card() << endl;
```

δίνουν:

```
ABC
ACDE
ABCDE
ABCDE
5
```

✍✍✍

Prj05.3.3 Διαφορά Συνόλων – Διαγραφή Στοιχείου Συνόλου

Αφού η διαφορά συνόλων $x \setminus y$ είναι το σύνολο των στοιχείων του x που δεν ανήκουν στο y, η διαφορά σχετίζεται με τη διαγραφή στοιχείου συνόλου όπως σχετίζεται η ένωση με την εισαγωγή:

`true` { διαγραφή του u από το x } $u \notin x$

`true` { $x = x \setminus \{ u \}$ } $u \notin x$

Θα πρέπει λοιπόν να ξεκινήσουμε και εδώ με την επιφόρτωση του "`-=`" με μια μέθοδο που θα υλοποιεί την "`\=`". Να σημειώσουμε ότι καταφεύγουμε στον "`-`" που χρησιμοποιεί η Pascal και αγνοούμε τον "~" που χρησιμοποιείται στη διατύπωση του προβλήματος διότι ο "~" δεν είναι κατάλληλος: είναι *ενικός*!

Μπορούμε να δούμε τη διαφορά $x \setminus y$ ως την τομή, με το x, του συνόλου των στοιχείων που ανήκουν στο x ή στο y (αλλά όχι και στα δύο):

```
SetOfUCL& SetOfUCL::operator-=( const SetOfUCL& rhs )
{
   bitmap = bitmap & ( bitmap ^ rhs.bitmap );
   return *this;
} // SetOfUCL::operator-=
```

και για όποιον προτιμάει συνάρτηση με όνομα:

```
SetOfUCL& SetOfUCL::setDifference( const SetOfUCL& rhs )
{
   return ( *this -= rhs );
} // SetOfUCL::setDifference
```

Από αυτήν παίρνουμε την καθολική συνάρτηση με τη πάγια τεχνική:

```
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv -= rhs;
   return fv;
} // operator-( SetOfUCL, char )
```

και τη μέθοδο διαγραφής:

```
SetOfUCL& SetOfUCL::operator-=( char c )
{
   if ( isupper(c) )
   {
      *this -= SetOfUCL( c );
   }
   return *this;
} // SetOfUCL::operator-=
```

Πρόσεξε ότι εδώ *δεν* ρίχνουμε εξαίρεση αν το c δεν είναι κεφαλαίο λατινικό γράμμα· είναι σίγουρο ότι –και στην περίπτωση αυτή– μετά την εκτέλεση της πράξης $c \notin$ `*this`.

Η μέθοδος

```
SetOfUCL& SetOfUCL::remove( char c )
{
   return ( *this -= c );
} // SetOfUCL::remove
```

κάνει την ίδια δουλειά.

Παράδειγμα

Οι εντολές

```
   x.display( cout );  y.display( cout );
   y -= x;
   y.display( cout );
   y -= 'D';
   y.display( cout );
   y -= 'Q';
   y.display( cout );
   z = x - y;
   z.display( cout );
```

δίνουν:

```
ABC
ACDE
DE
```

```
E
E
ABC
```

Prj05.3.4 Τομή Συνόλων: $x \cap= y$

Ακολουθώντας την Pascal, επιφορτώνουμε τον "`*=`" για την υλοποίηση της πράξης "$\cap$=":

```
SetOfUCL& SetOfUCL::operator*=( const SetOfUCL& rhs )
{
   bitmap = bitmap & rhs.bitmap;
   return *this;
} // SetOfUCL::operator*=
```

Από αυτήν παίρνουμε την καθολική συνάρτηση:

```
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv *= rhs;
   return fv;
} // operator*( SetOfUCL, SetOfUCL )
```

Prj05.4 Το Πρόγραμμα

Ας ξεκινήσουμε με την παρατήρηση ότι: *από κάθε παράγραφο χρειαζόμαστε μόνον το σύνολο των γραμμάτων της* (αφού τα μετατρέψουμε σε κεφαλαία). Έχει νόημα λοιπόν να γράψουμε μια συνάρτηση

```
void readParagraph( ifstream& tin, SetOfUCL& letSet )
```

που θα διαβάζει μέσω του ρεύματος *tin* μια παράγραφο από ένα αρχείο text και θα μας δίνει το σύνολο *letSet* των γραμμάτων που έχει.

Πώς θα δουλεύει η συνάρτηση; Έτσι περίπου:

```
"Άδειασε" το letSet
Διάβασε έναν χαρακτήρα c
while ( δεν τελείωσε η παράγραφος )
{
   if ( ο c είναι γράμμα )
      Βάλε στο letSet το αντίστοιχο κεφαλαίο
   Διάβασε έναν χαρακτήρα c
}
```

Πώς αδειάζουμε το *letSet*; Θα πρέπει να μηδενίσουμε το *bitmap* της *letSet*. Θα πρέπει να εφοδιάσουμε την κλάση μας με μια κατάλληλη μέθοδο[3]:

```
   void clear()  {  bitmap = 0L; };
```

οπότε η "*«Άδειασε» το letSet*" γίνεται:[4]

```
   letSet.clear();
```

Διαβάζουμε τον *c* με την:

```
   tin.get(c);
```

Η "*Βάλε στο letSet το αντίστοιχο κεφαλαίο*" γίνεται:

```
   letSet.insert( toupper(c) );
```

Ας έλθουμε τώρα στις συνθήκες. Η "*ο c είναι γράμμα*" είναι απλή: `isalpha(c)`.

---

[3] Για να μπορούμε να ελέγχουμε αν ένα σύνολο είναι *κενό* γράφουμε μια:

```
bool isEmpty() const {  return ( bitmap == 0UL );  }
```

[4] Η *clear()* δεν είναι απαραίτητη. Αφού ο ερήμην δημιουργός δημιουργεί ένα κενό σύνολο, μπορούμε να «αδειάσουμε» το *letSet* γράφοντας απλώς: "`letSet = SetOfUCL()`".

Η *"δεν τελείωσε η παράγραφος"* θέλει λίγη σκέψη. Πότε τελειώνει η παράγραφος;

- Όταν βρούμε "`'\n''\n'`" ή
- όταν βρούμε τέλος αρχείου.

Για να μπορούμε να ανιχνεύουμε το "`'\n''\n'`" θα πρέπει να θυμόμαστε τον προηγούμενο χαρακτήρα ή να διαβάζουμε προκαταβολικά (*peek*) τον επόμενο. Ας προτιμήσουμε την πρώτη λύση που δεν αλλάζει και την κανονική ροή της ανάγνωσης. Δηλώνουμε:

```
char c, lastC;
```

και στη *lastC* θα κρατούμε την προηγούμενη τιμή της *c*:

```
lastC = c;  tin.get(c);
```

Έτσι, θα μπορούμε να ελέγξουμε:

```
if ( c == '\n' )
{
   eopar = ( lastC == '\n' );
}
```

Προηγουμένως έχουμε δηλώσει:

```
bool eopar;     // end of paragraph
```

Με βάση τα παραπάνω η *"δεν τελείωσε η παράγραφος"* γίνεται:

```
!tin.eof() && !eopar
```

Αρχικώς βάζουμε:

```
eopar = false;  lastC = '\0';
```

Να λοιπόν η συνάρτηση:

```
void readParagraph( ifstream& tin, SetOfUCL& letSet )
{
   bool eopar;     // end of paragraph
   char c, lastC;

   letSet.clear();
   eopar = false;  lastC = '\0';
   tin.get( c );
   while ( !tin.eof() && !eopar )
   {
      if ( isalpha( c ) )
      {
         letSet.insert( toupper( c ) );
      }
      else if ( c == '\n' )
      {
         eopar = ( lastC == '\n' );
      }
      lastC = c;  tin.get( c );
   } // while
} // readParagraph
```

Τώρα το πρόγραμμά μας γίνεται πιο εύκολο. Ας προσπαθήσουμε να κάνουμε ένα σχέδιο:

```
Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της
Διάβασε τη δεύτερη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

Τώρα θα πρέπει να ξανακάνουμε τα ίδια αλλά το ρόλο της πρώτης παραγράφου θα τον παίζει η δεύτερη και της δεύτερης η τρίτη. Έχουμε όμως μια διαφορά: τη δεύτερη παράγραφο την έχουμε διαβάσει ήδη και τα γράμματά της υπάρχουν στο *nextSet*. Θα δουλέψουμε λοιπόν ως εξής:

```
Βάλε curSet = nextSet
Διάβασε την τρίτη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

```
Βάλε curSet = nextSet
Διάβασε την τέταρτη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
. . .
```

Όταν τελειώσει αυτή η διαδικασία, διότι θα βρούμε τέλος αρχείου, θα έχουμε στο *curSet* το σύνολο των γραμμάτων της τελευταίας παραγράφου. Τι θα κάνουμε με αυτό; Το πιο φυσιολογικό είναι να θεωρήσουμε ότι υπάρχει και μια «κενή παράγραφος» (με κενό σύνολο γραμμάτων) μετά από αυτήν.

Να λοιπόν τι θα πρέπει να κάνουμε:

```
Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της
while ( !tin.eof() )
{
   Διάβασε την επόμενη παράγραφο και πάρε το σύνολο nextSet των
   γραμμάτων της
   Υπολόγισε τα σύνολα
   common = curSet ∩ nextSet και notInNext = curSet \ nextSet
   Γράψε στο αρχείο τα curSet, common, notInNext
   Βάλε curSet = nextSet
}
Βάλε nextSet = { }
Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet
Γράψε στο αρχείο τα curSet, common, notInNext
```

Ας γράψουμε το πρόγραμμά μας. Στην αρχή δηλώνουμε:

```
   SetOfUCL curSet, nextSet, common, notInNext;
   ifstream tin( "mstrpc.txt" );
   ofstream bout( "paragltr.dta", ios::binary );
```

και αρχίζουμε τη μετάφραση:

Η «*Διάβασε την πρώτη παράγραφο και πάρε το σύνολο curSet των γραμμάτων της*» μεταφράζεται στην:

```
   readParagraph( tin, curSet );
```

Παρομοίως η «*Διάβασε την επόμενη παράγραφο και πάρε το σύνολο nextSet των γραμμάτων της*» γίνεται:

```
   readParagraph( tin, nextSet );
```

Η «*Υπολόγισε τα σύνολα common = curSet ∩ nextSet και notInNext = curSet \ nextSet*« γίνεται:

```
      common = curSet * nextSet;
      notInNext = curSet - nextSet;
```

Και με την «*Γράψε στο αρχείο τα curSet, common, notInNext*» τι κάνουμε;

Ένας τρόπος να την υλοποιήσουμε είναι να γράψουμε:

```
      bout.write( reinterpret_cast<char*>( &curSet ),
                  sizeof( curSet ) );
```

και στη συνέχεια τα παρόμοια για τα άλλα δύο σύνολα.

Ένας άλλος είναι να εφοδιάσουμε την κλάση μας με μια μέθοδο:

```
void SetOfUCL::save( ostream& bout ) const
{
   bout.write( reinterpret_cast<const char*>(&bitmap), sizeof(bitmap) );
} // SetOfUCL::save
```

και να έχουμε το πρόγραμμά μας πιο απλό:

```
      curSet.save( bout );
      common.save( bout );  notInNext.save( bout );
```

Να λοιπόν το πρόγραμμά μας:

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

```
#include "SetOfUCL.cpp"

using namespace std;

void readParagraph( ifstream& tin, SetOfUCL& letSet );

int main()
{
   SetOfUCL curSet, nextSet, common, notInNext;
   ifstream tin( "mstrpc.txt" );
   ofstream bout( "paragltr.dta", ios::binary );

   readParagraph( tin, curSet );
   while ( !tin.eof() )
   {
      readParagraph( tin, nextSet );
      common = curSet * nextSet;
      notInNext = curSet - nextSet;
      curSet.save( bout );
      common.save( bout );  notInNext.save( bout );
      curSet = nextSet;
   } // while
   tin.close();
   nextSet.clear();
   common = curSet * nextSet;
   notInNext = curSet - nextSet;
   curSet.save( bout );
   common.save( bout );  notInNext.save( bout );
   bout.close();
} // main

void readParagraph( ifstream& tin, SetOfUCL& letSet )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Prj05.4.1 Και η *SetOfUCL* Μέχρι Τώρα

Λύσαμε το πρόβλημά μας έχοντας την εξής κλάση:

```
class SetOfUCL
{
public:
   SetOfUCL() {  bitmap = 0UL;  };
   explicit SetOfUCL( char c );
   ~SetOfUCL() { };
   unsigned int card() const;
   SetOfUCL& operator+=( const SetOfUCL& rhs );
   SetOfUCL& setUnion( const SetOfUCL& rhs );
   SetOfUCL& operator+=( char c );
   SetOfUCL& insert( char c );
   SetOfUCL& operator-=( const SetOfUCL& rhs );
   SetOfUCL& setDifference( const SetOfUCL& rhs );
   SetOfUCL& operator-=( char c );
   SetOfUCL& remove( char c );
   SetOfUCL& operator*=( const SetOfUCL& rhs );
   SetOfUCL& setIntersection( const SetOfUCL& rhs );
   void clear()  {  bitmap = 0UL; };
   bool isEmpty() const  {  return ( bitmap == 0UL );  };
   void save( ostream& bout ) const;
private:
   unsigned long int bitmap;
}; // SetOfUCL

SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator+( const SetOfUCL& lhs, char rhs );
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs );
```

```
SetOfUCL operator-( const SetOfUCL& lhs, char rhs );
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs );
```

Prj05.5 Εμπλουτίζοντας την Κλάση

Συγκρίνοντας τα παραπάνω με αυτά που ξέρουμε από τη Θεωρία Συνόλων βλέπουμε ότι μας λείπουν δύο πάγιες διμελείς σχέσεις των συνόλων: η σχέση "*ανήκειν*":[5]

$_ \in _ : X \leftrightarrow \mathbb{P}\, X$

και η σχέση "*περιέχεσθαι*":

$_ \subseteq _ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$

όπου:

$\forall x, y: \mathbb{P}\, X \bullet (x \subseteq y \Leftrightarrow \forall u{:}X \bullet u \in x \Rightarrow u \in y)$

Ας ξεκινήσουμε από την "∈". Υπάρχει κάποιος τελεστής για να επιφορτώσουμε; Η C++ δεν έχει "`in`" και δεν φαίνεται να ταιριάζει κάποιος άλλος τελεστής. Θα πρέπει να χρησιμοποιήσουμε κάποιο όνομα, π.χ.:

"*u* `isMemberOf` *x*" ή "*x* `hasMember` *u*"

Αν θέλουμε να γράψουμε μια μέθοδο καλύτερα να χρησιμοποιήσουμε το δεύτερο αφού για την "$u \in x$" θα γράφουμε "`x.hasMember(u)`".

Θα υλοποιήσουμε τη μέθοδο που θα επιστρέφει τιμή τύπου *bool* –αφού είναι κατηγόρημα– και θα γράψουμε προσαρμόζοντας την *bitValue()*:

```
bool SetOfUCL::hasMember( char c ) const
{
   bool fv( false );

   if ( isupper( c ) )
   {
      unsigned long int x( 1UL );
      x <<= static_cast<int>(c) - static_cast<int>('A');
      fv = ( ( bitmap & x ) != 0 );
   }
   return fv;
} // SetOfUCL::hasMember
```

Πρόσεξε ότι όταν ο *c* δεν είναι κεφαλαίο γράμμα δεν ρίχνουμε εξαίρεση· λέμε απλώς ότι δεν ανήκει στο σύνολο.

Αν προτιμάς το πρώτο όνομα, καλύτερα γράψε μια καθολική συνάρτηση:

```
bool isMemberOf( char c, const SetOfUCL& rhs )
```

που θα την καλείς ως εξής: "`isMemberOf( u, x )`". Φυσικά, αυτήν θα πρέπει να δηλώσεις ως **friend** της κλάσης.[6]

Για την υλοποίηση της δεύτερης δεν μας διευκολύνει και τόσο το κατηγόρημα που δίνουμε παραπάνω. Η συνάρτηση που θα προκύψει θα είναι πολύ αργή. Ένας άλλος τρόπος είναι να στηριχθούμε στην ιδιότητα:

$$(x \subseteq y) \Leftrightarrow (x = x \cap y)$$

Έχουμε λοιπόν:

```
bool SetOfUCL::isSubset( SetOfUCL rhs ) const
{  return ( bitmap == ( bitmap & rhs.bitmap ) );  } // isSubset
```

Αν *x*, *y* είναι *SetOfUCL* τότε η συνθήκη "$x \subseteq y$" θα «μεταφράζεται» στο πρόγραμμά μας σε "`x.isSubset(y)`".

---

[5] Θυμίσου ότι κάθε σύνολο με στοιχεία από το (*σύνολο βάσης*) *X* είναι μέλος του *δυναμοσυνόλου* $\mathbb{P}\, X$ του *X*.

[6] Βέβαια, αν δεν φοβάσαι μπερδέματα, μπορείς να έχεις και τις δύο, αρκεί να ορίσεις:

```
bool isMemberOf( char c, const SetOfUCL& rhs )  {  return rhs.hasMember( c );  }
```

Η Pascal χρησιμοποιεί τον "`<=`" για το υποσύνολο. Μπορούμε λοιπόν να τον επιφορτώσουμε ως καθολική συνάρτηση:

```
bool operator<=( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return lhs.isSubset( rhs );
} // operator<=( SetOfUCL...
```

και αντί για "`x.isSubset(y)`" να γράφουμε "`x <= y`".

Αφού η Pascal χρησιμοποιεί και τον "`>=`" για το υπερσύνολο, μπορούμε να τον επιφορτώσουμε χρησιμοποιώντας την *isSubset()*:

```
bool operator>=( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return rhs.isSubset( lhs );
} // operator<=( SetOfUCL...
```

Δεν πρέπει να παραλείψουμε να επιφορτώσουμε τον "`==`" για να μπορούμε να συγκρίνουμε δύο σύνολα για ισότητα:

```
bool operator==( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return ( lhs.bitmap == rhs.bitmap );
} // operator==( const SetOfUCL
```

και

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

Πρόσεξε ότι η πρώτη θα πρέπει να δηλωθεί στην κλάση ως **friend**.[7]

Αν θέλουμε να έχουμε τη δυνατότητα να ελέγχουμε για γνήσιο *υποσύνολο* (proper subset) γράφουμε:

```
bool SetOfUCL::isProperSubset( const SetOfUCL& rhs ) const
{
   return ( this->isSubset(rhs) && (bitmap != rhs.bitmap) );
} // isProperSubset
```

και επιφορτώνουμε:

```
bool operator<( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   return lhs.isProperSubset( rhs );
} // SetOfUCL::operator<
```

Παρατήρηση: ▶
Εδώ προσοχή: *Η "`x < y`" και η "`x >= y`" δεν έχουν την παραμικρή σχέση!* Το ότι το *x* δεν είναι γνήσιο υποσύνολο του *y* –δηλαδή δεν ισχύει η "`x < y`"– δεν σημαίνει ότι το *x* είναι υπερσύνολο του *y*. Μάλλον η ιδέα να επιφορτώσουμε τον "`<`" δεν είναι και τόσο καλή! ◀

Τελικώς, το περιεχόμενο του `SetOfUCL.h` έχει γίνει:

```
#ifndef _SETOFUCL_H
#define _SETOFUCL_H

#include <string>
#include <fstream>

using namespace std;

class SetOfUCL
{
friend bool operator==( const SetOfUCL& lhs, const SetOfUCL& rhs );
public:
   SetOfUCL() {  bitmap = 0UL;  };
```

[7] Να γράψουμε "`return ( lhs.isSubset(rhs) && rhs.isSubset(lhs) )`" για να αποφύγουμε το **friend**; Ε, όχι και έτσι...

```
   explicit SetOfUCL( char c );
   ~SetOfUCL() { };
   unsigned int card() const;
   SetOfUCL& operator+=( const SetOfUCL& rhs );
   SetOfUCL& setUnion( const SetOfUCL& rhs );
   SetOfUCL& operator+=( char c );
   SetOfUCL& insert( char c );
   SetOfUCL& operator-=( const SetOfUCL& rhs );
   SetOfUCL& setDifference( const SetOfUCL& rhs );
   SetOfUCL& operator-=( char c );
   SetOfUCL& remove( char c );
   SetOfUCL& operator*=( const SetOfUCL& rhs );
   SetOfUCL& setIntersection( const SetOfUCL& rhs );
   void clear()  {  bitmap = 0UL; };
   bool isEmpty() const  {  return ( bitmap == 0UL );  };
   void save( ostream& bout ) const;

   bool hasMember( char c ) const;
   bool isSubset( SetOfUCL rhs ) const;
   bool isProperSubset( const SetOfUCL& rhs ) const;

   void display( ostream& tout ) const;
private:
   unsigned long int bitmap;
}; // SetOfUCL

SetOfUCL operator+( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator+( const SetOfUCL& lhs, char rhs );
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs );
SetOfUCL operator-( const SetOfUCL& lhs, char rhs );
SetOfUCL operator*( const SetOfUCL& lhs, const SetOfUCL& rhs );

bool operator<=( const SetOfUCL& lhs, const SetOfUCL& rhs );
bool operator<( const SetOfUCL& lhs, const SetOfUCL& rhs );
bool operator!=( const SetOfUCL& lhs, const SetOfUCL& rhs );

struct SetOfUCLXptn
{
   enum { nonUCL };
   char funcName[100];
   int  errorCode;
   char errCharVal;
   SetOfUCLXptn( const char* mn, int ec, char c=0 )
      :  errorCode( ec ),  errCharVal( c )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // SetOfUCLXptn
```

Prj05.6 Σχόλια και Παρατηρήσεις

Ας επαναλάβουμε και εδώ μερικές παρατηρήσεις σχετικώς με τις διάφορες επιλογές που κάναμε αναπτύσσοντας την κλάση.

- Για την ένωση και τη διαφορά ορίσαμε (επιφορτώσαμε) τους "+=" και "-=" και από αυτούς πήραμε όλους τους άλλους τελεστές (και σχετικές μεθόδους) σχεδόν με «μηχανικό τρόπο». Έτσι έχουμε εξασφαλισμένο ότι ο "+" έχει παντού το ίδιο νόημα (το ίδιο ισχύει και για τον "-"). Αν κάτι πρέπει να αλλάξει, αυτό θα γίνει σε ένα σημείο μόνο.
- Η τεχνική που μάθαμε στην §22.7.1 και εφαρμόσαμε για να πάμε από τον "+=" στον "+" και από τον "-=" στον "-" έχει έναν περιορισμό: δεν μπορεί να επιστρέψει τύπο αναφοράς. Πρόσεξε την περίπτωση με τη διαφορά:

```
SetOfUCL operator-( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv -= rhs;
```

```
   return fv;
} // operator-( SetOfUCL, SetOfUCL )
```

Όταν κληθεί η `operator-()` δημιουργείται η *fv* που

- είναι τοπική στη συνάρτηση και
- ζει όσο εκτελείται η συνάρτηση.

Όταν έλθει η στιγμή να χρησιμοποιηθεί η τιμή που επιστρέφει –για παράδειγμα εκτελεσθεί η εκχώρηση στη *notInNext* στην "`notInNext = curSet - nextSet`"– η εκτέλεση της `operator-()` έχει τελειώσει, η *fv* δεν υπάρχει πια και η συνάρτηση επιστρέφει ένα βέλος προς μια ανύπαρκτη μεταβλητή.

Τι θα μπορούσαμε να κάνουμε; Να κάνουμε την *fv* δυναμική και να γράψουμε τη συνάρτηση έτσι ώστε να επιστρέφει βέλος προς αυτήν.

- Κατά τα άλλα η τεχνική αυτή δεν είναι μόνο για τελεστές. Ας πούμε ότι έχουμε την

```
SetOfUCL& SetOfUCL::setDifference( const SetOfUCL& rhs )
{
   return ( *this -= rhs );
} // SetOfUCL::setDifference
```

και θέλουμε να ορίσουμε και την καθολική συνάρτηση *setDifference()* που θα κάνει την ίδια δουλειά με την `operator-()`. Γράφουμε:

```
SetOfUCL setDifference( const SetOfUCL& lhs, const SetOfUCL& rhs )
{
   SetOfUCL fv( lhs );
   fv.setDifference( rhs );
   return fv;
} // setDifference( SetOfUCL, SetOfUCL )
```

κεφάλαιο

23

Κληρονομιές

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα γνωρίσουμε την έννοια της παραγωγής μιας κλάσης από μια άλλη και της κληρονομιάς που παίρνει η νέα από την παλιά.
Προσδοκώμενα αποτελέσματα:
Εξοικείωση με την κλήρονομικότητα και τις σχετικές προγραμματιστικές τεχνικές.
Έννοιες κλειδιά:

- *βασική κλάση*
- *παράγωγη κλάση*
- *σχέση* `is_a`
- *προστατευόμενα (*`protected`*) μέλη*
- *εικονικές (*`virtual`*) μέθοδοι*
- *πολυμορφισμός*
- *δυναμική τυποθεώρηση*
- *εξακρίβωση τύπου κατά την εκτέλεση -- RTTI*

Περιεχόμενα:

23.1 Κτίζοντας Πάνω στα Υπάρχοντα ... 842
23.1.1 Τι ΔΕΝ Κληρονομείται ... 843
23.2 Σχέσεις Αντικειμένων και Κλάσεων ... 844
23.3 Ο Νέος Δημιουργός ... 846
23.3.1 Και Ένας Άλλος Τρόπος ... 848
23.3.2 Ο Δημιουργός Αντιγραφής ... 848
23.4 Και ο Καταστροφέας ... 849
23.5 Νέες και Παλιές Μέθοδοι ... 849
23.5.1 Ο Τελεστής Εκχώρησης ... 852
23.6 Καθολικές Συναρτήσεις ... 853
23.7 Κλάση Εξαιρέσεων Παράγωγης Κλάσης ... 853
23.8 "`private`" ή "`protected`" ... 854
23.8.1 * "`protected`": Ψιλά Γράμματα ... 855
23.9 Εικονικές Μέθοδοι ... 857
23.10 * Υλοποίηση Εικονικών Συναρτήσεων ... 859
23.10.1 "`virtual`" και Τεμαχισμός ... 860
23.10.2 Κάποια Σχόλια... ... 861
23.11 Εικονικός Καταστροφέας ... 861
23.12 Περί Πολυμορφισμού ... 862
23.12.1 Πολυμορφισμός Χρόνου Μεταγλώττισης(;) ... 863
23.12.2 * Υπερίσχυση ή Επιφόρτωση ... 864
23.13 Δυναμική Τυποθεώρηση - RTTI ... 866
23.13.1 Μετατροπή Αναφορών ... 868
23.14 "`is_a`" ή "`has_a`" ... 869

23.15 "private", "protected" και "public" 872
23.16 Αφηρημένες Κλάσεις 874
23.17 Η Σειρά Δημιουργίας 875
23.18 Πολλαπλή Κληρονομιά 876
23.19 Διεπαφές 878
23.20 Ανακεφαλαίωση 879
Ερωτήσεις - Ασκήσεις 879
Α Ομάδα 879
Β Ομάδα 880

Εισαγωγικές Παρατηρήσεις:
Ένα βασικό χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού είναι η κληρονομικότητα: η δυνατότητα μιας κλάσης να δηλώνεται με βάση μιαν άλλη κλάση που προϋπάρχει και να έχει ως **κληρονομιά** (inheritance) τα μέλη και τις μεθόδους της αρχικής.

Στη συνέχεια θα αποφύγουμε τις πολλές θεωρίες και δούμε τα περί κληρονομιάς με παραδείγματα σε C++ (αφού, καλώς ή κακώς, οι τρόποι υλοποίησης των βασικών εννοιών στη C++ έχουν το ίδιο μεγάλο ενδιαφέρον με τις ίδιες τις έννοιες.)

23.1 Κτίζοντας Πάνω στα Υπάρχοντα

Η ημερομηνία ως χρονικός προσδιορισμός δεν είναι πάντοτε επαρκής. Οι περιπτώσεις που χρειαζόμαστε μεγαλύτερη ακρίβεια είναι πολλές. Αφού η *Date* δεν επαρκεί ας γράψουμε μια νέα κλάση, ας την πούμε *DateTime*, που θα πηγαίνει μέχρι δευτερόλεπτο:

```
class DateTime
{
public:
// . . .
private:
   unsigned int  dtYear;
   unsigned char dtMonth;
   unsigned char dtDay;
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Τι θα κάνουμε με τα *dtYear*, *dtMonth*, *dtDay*; Αυτό μπορούμε να το αντιγράψουμε από τη *Date*, αλλά δεν χρειάζεται! Δες έναν άλλον τρόπο να γράψουμε τα παραπάνω με πολύ «περισσότερο περιεχόμενο»:

```
class DateTime: public Date
{
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
// . . .
private:
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Αφήνοντας κατά μέρος τον δημιουργό, πού βρίσκεται το «περισσότερο περιεχόμενο»; Μας το δείχνει το παρακάτω προγραμματάκι:

```
   DateTime dt( 2007, 11, 26, 19, 30, 31 );
   cout << dt.getDay() << '.' << dt.getMonth() << '.' << dt.getYear() << endl;
   cout << dt << endl;
   dt += 3;          cout << dt << endl;
   dt.forward( 3 );  cout << dt << endl;
   ++dt;             cout << dt << endl;
   cout << (dt < Date(2008)) << endl;
```

που μας δίνουν:

```
26.11.2007
26.11.2007
26.11.2007
26.11.2007
27.11.2007
1
```

Η *dt* δηλώθηκε μεν κλάσης *DateTime* αλλά συμπεριφέρεται πλήρως ως αντικείμενο κλάσης *Date*! Ή αλλιώς:

- Η κλάση *DateTime* **κληρονομεί** (inherits) –για κάθε αντικείμενό της– όλα τα χαρακτηριστικά και τη συμπεριφορά που έχει ένα αντικείμενο της *Date*.
- Αλλά, κάθε αντικείμενό της *DateTime* έχει επιπλέον χαρακτηριστικά και μπορούμε να «εμπλουτίσουμε» τη συμπεριφορά του με επιπλέον μεθόδους. Αυτό και θα κάνουμε στη συνέχεια.

Αφού η *DateTime* κληρονομεί τη συμπεριφορά της *Date* θα κληρονομεί και την αναλλοίωτη. Πράγματι, ας δούμε την αναλλοίωτη της νέας κλάσης:

$$I_{DateTime}: (dYear > 0) \;\&\&\; (0 < dMonth <= 12) \;\&\&\; (0 < dDay <= lastDay(dYear, dMonth)) \;\&\&\; (0 < dtHour < 24) \;\&\&\; (0 < dtMin < 60) \;\&\&\; (0 < dtSec < 60)$$

Ή

$$I_{DateTime}: I_{Date} \;\&\&\; (0 < dtHour < 24) \;\&\&\; (0 < dtMin < 60) \;\&\&\; (0 < dtSec < 60)$$

Δηλαδή, αφού η I_{Date} συνδέεται με τα υπόλοιπα με "&&":

♦ ***Ό,τι ισχύει για τα αντικείμενα της βασικής κλάσης ισχύει και για τα αντικείμενα της παράγωγης κλάσης.***

Αυτή είναι η **αρχή της Liskov** (Liskov & Wing, 1993).

«*Βασική*»; «*Παράγωγη*»; Ήλθε η ώρα για λίγη ορολογία: Λέμε ότι η κλάση *DateTime* **παράγεται** (is derived) από την *Date* και παίρνει ως **κληρονομιά** (inheritance) όλα τα μέλη και τις μεθόδους της. Λέμε ότι η (κληρονομούμενη) κλάση *Date* είναι η

- **βασική κλάση** (base class) ή
- **υπερκλάση** (superclass) ή
- **γονική** (parent) κλάση

ενώ η κληρονόμος *DateTime* ονομάζεται

- **παράγωγη** (derived) κλάση ή
- **υποκλάση** (subclass) ή
- **κλάση-παιδί** (child class).

23.1.1 Τι ΔΕΝ Κληρονομείται

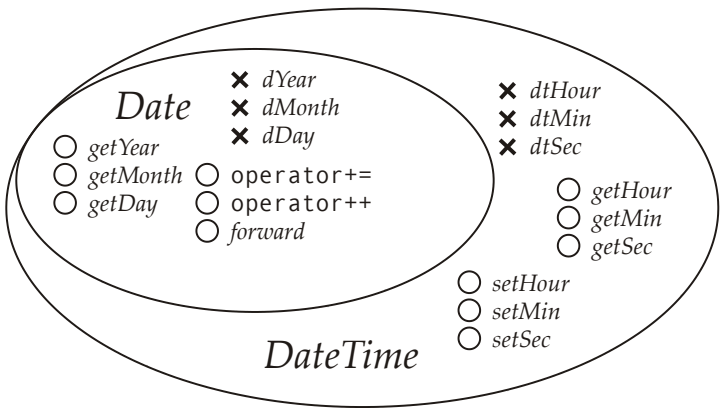
Γενικώς:

♦ ***Οι δημιουργοί δεν κληρονομούνται.***

με την έννοια που είδαμε στο παράδειγμα. Πέρα από αυτό όμως, υπάρχουν τέσσερις «ειδικές συναρτήσεις»:

- ο ερήμην δημιουργός,
- ο δημιουργός αντιγραφής,
- ο τελεστής εκχώρησης,
- ο καταστροφέας,

που όχι μόνο δεν κληρονομούνται με την έννοια που είδαμε πιο πάνω, αλλά αν δεν ορίσουμε κάποια από αυτές, όταν ορίζουμε μια κλάση, ο μεταγλωττιστής θα ορίσει αυτομάτως μια *συναγόμενη* (implicit). Αυτό, όπως ξέρουμε, δεν σημαίνει ότι η συναγόμενη συνάρτηση



Σχ. 23-1 Κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

θα συμπεριφέρεται όπως τη θέλουμε. Για τον λόγο αυτόν ακριβώς έχουμε βάλει κανόνες για το πότε πρέπει να παρεμβαίνουμε.

Τα παραπάνω ισχύουν και για τις παράγωγες κλάσεις: *Αν δεν ορίσουμε οποιαδήποτε από τις παραπάνω συναρτήσεις ο μεταγλωττιστής θα ορίσει αυτομάτως μια συναγόμενη.*

Αν όμως θελήσουμε να ορίσουμε δικές μας συναρτήσεις, μπορούμε να χρησιμοποιήσουμε τις αντίστοιχες της βασικής κλάσης όπως θα δούμε στη συνέχεια.

Τέλος, να πούμε ότι:

- ***Οι φιλίες δεν κληρονομούνται.***

Δηλαδή, το ότι μια συνάρτηση ή μια κλάση είναι φίλη της βασικής κλάσης δεν σημαίνει ότι αυτομάτως είναι και φίλη της παράγωγης. Αν θέλεις να είναι φίλη της παράγωγης θα πρέπει να το δηλώσεις.

23.2 Σχέσεις Αντικειμένων και Κλάσεων

Δες το Σχ. 23-1: να πώς περίπου είναι ένα αντικείμενο κλάσης *DateTime*. Όπως βλέπεις περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

Πώς ερμηνεύονται λοιπόν αυτά που είδαμε στο προηγούμενο παράδειγμα; Δηλαδή: γιατί ένα αντικείμενο κλάσης *DateTime* συμπεριφέρεται σαν αντικείμενο κλάσης *Date*; Διότι κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date* που ανταποκρίθηκε με τον τρόπο που είδαμε σε όλα τα μηνύματα που του στείλαμε.

Γενικεύοντας δίνουμε δύο κανόνες:

- ***Κάθε αντικείμενο της παράγωγης κλάσης D περιέχει ένα υποαντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

  Λόγω αυτού του γεγονότος:

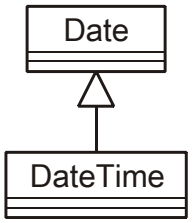
- ***Κάθε αντικείμενο της παράγωγης κλάσης D συμπεριφέρεται σαν αντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

Και ακόμα γενικότερα:

- ***Οπουδήποτε, μέσα στο πρόγραμμά μας μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D.***

Αυτό λέγεται **δυνατότητα υποκατάστασης** (substitutability) και είναι μια άλλη θεώρηση της αρχής της Liskov (Liskov Substitution Principle - LSP).[1]

[1] Στο τέλος του κεφαλαίου θα δεις μια ακριβέστερη διατύπωση.



Σχ. 23-2 Παράσταση της συσχέτισης *DateTime* `is_a` *Date*.

Για παράδειγμα, αν έχουμε δηλώσει:

```
Date d;
DateTime dt;
Date* pD;
DateTime* pDt;
```

είναι νόμιμες οι εντολές:

```
d = dt;        // στη d εκχωρείται
               // το υποαντικείμενο κλάσης Date της dt
pD = new DateTime;
```

αλλά δεν είναι νόμιμες οι:[2]

```
dt = d;
pDt = new Date;
```

Το ότι με τη "`d = dt`" εκχωρείται στη *d* το υποαντικείμενο κλάσης *Date* της *dt* ονομάζεται **τεμαχισμός** (slicing) του αντικειμένου. Αυτό θα το ξαναδούμε...

Πρόσεξε όμως και το εξής: Αν δούμε τις κλάσεις ως σύνολα αντικειμένων τότε *η βασική κλάση B περιέχει και τα αντικείμενα της παράγωγης κλάσης D*, δηλαδή η *D* είναι υποσύνολο της *B* (ενώ το αντικείμενο της *B* είναι «υποσύνολο» του αντικειμένου της *D*). Αντί για $D \subseteq B$ γράφουμε *D* `is_a` *B* και δίνουμε μια διαγραμματική παράσταση σαν αυτήν του Σχ. 23-2. Το βέλος δείχνει από την παράγωγη προς τη βασική. Το νόημά του είναι «*η D **αναφέρεται προς** (refers to) την B*».

Σημείωση:▶
Ορισμένοι αντί για *D* `is_a` *B* γράφουν *D* `a_kind_of` *B* και κρατούν την `is_a` ως *σχέση μεταξύ των αντικειμένων* των δύο κλάσεων. Εμείς χρησιμοποιούμε το `is_a` και για τις δύο περιπτώσεις.◀

Στη συνέχεια θα δούμε πώς μπορούμε να χειριζόμαστε το υποαντικείμενο της βασικής κλάσης μέσα στις μεθόδους της παράγωγης. Προς το παρόν να πούμε ότι μπορείς να φτάσεις σε αυτό ως εξής:

- Το `this` είναι βέλος προς αντικείμενο την παράγωγης κλάσης, στο παράδειγμά μας τύπου *DateTime**.
- Η παράσταση "`static_cast<Date*>(this)`" μας δίνει ένα αντίγραφο του `this` αλλά τύπου *Date**. Αυτό *είναι βέλος προς το υποαντικείμενο κλάσης Date**.
- Το "`*(static_cast<Date*>(this))`" είναι *το υποαντικείμενο κλάσης Date*.

Να και ένα σχετικό

Παράδειγμα✍
Η "`dt = d`" απαγορεύεται με την έννοια ότι δεν υπάρχει παρόμοια *συναγόμενη* επιφόρτωση του "`=`". Φυσικά, αν την ορίσουμε δεν υπάρχει πρόβλημα.

Αν ορίσουμε, για παράδειγμα τον *δημιουργό μετατροπής*:

```
DateTime& DateTime::operator=( const Date& rhs )
{
   *(static_cast<Date*>(this)) = rhs:
   dtHour = 12;  dtMin = 0;  dtSec = 0;
} // DateTime::operator=
```

–που είναι κάτι λογικό– μια τέτοια εκχώρηση γίνεται νόμιμη.
✍✍✍

Έξω από το αντικείμενο της παράγωγης κλάσης. μπορείς να πάρεις ένα αντίγραφο του υποαντικειμένου της βασικής με στατική τυποθεώρηση. Για παράδειγμα, το

"`static_cast<Date>(dt)`"

θα σου δώσει ένα αντίγραφο του υποαντικειμένου κλάσης *Date* του *dt*.

Η "`d = dt`", που είδαμε, εκτελείται ως "`d = static_cast<Date>(dt)`".

---

[2] Αργότερα θα δούμε πώς μπορούμε να κάνουμε νόμιμες και αυτές τις εντολές.

23.3 Ο Νέος Δημιουργός

Ας πούμε ότι, λόγω εκτέλεσης μιας δήλωσης, δημιουργείται αντικείμενο που αποτελείται από άλλα αντικείμενα είτε λόγω κληρονομιάς είτε λόγω δήλωσης μελών. Η σειρά δημιουργίας αντικειμένου που είδαμε στην §21.3 χρειάζεται συμπλήρωση[3]:

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται.
- Εκτελείται το σώμα του δημιουργού.

Παράδειγμα

Στο πρόγραμμα:

```
 0:   #include <iostream>
 1:   using namespace std;
 2:
 3:   class A
 4:   {
 5:      int ma;
 6:   public:
 7:      A( int ap = 0 )
 8:      {  ma = ap;  cout << "A object created" << endl;  }
 9:   }; // A
10:
11:   class B
12:   {
13:      int mb;
14:   public:
15:      B( int bp = 0 )
16:      {  mb = bp;  cout << "B object created" << endl;  }
17:   }; // B
18:
19:   class C
20:   {
21:      int mc;
22:   public:
23:      C( int cp = 0 )
24:      {  mc = cp;  cout << "C object created" << endl;  }
25:   }; // C
26:
27:   class D: public B
28:   {
29:      A da;
30:      C dc;
31:   public:
32:      D() {  cout << "D object created" << endl;  }
33:   }; // D
34:
35:   int main()
36:   {
37:      D d;
38:   }
```

βλέπουμε ότι η κλάση *D* (γρ. 27-33) είναι παράγωγη της *B* (γρ. 11-17) και έχει δύο μέλη, το πρώτο (γρ. 29) κλάσης *A* (γρ. 3-9), το δεύτερο (γρ. 30) κλάσης *C* (γρ. 19-25). Με βάση αυτά που είπαμε, για να δημιουργηθεί το αντικείμενο *d* (γρ. 37 στη `main`)

- Πρώτα θα κληθεί ο δημιουργός της *B*, για να δημιουργήσει το υποαντικείμενο κλάσης *B* που θα υπάρχει στο *d*.
- Μετά θα δημιουργηθούν τα μέλη *da* και *dc* από τους δημιουργούς των κλάσεων *A* και *C*.
- Τέλος, θα εκτελεσθεί το σώμα του δημιουργού της *D*.

  Αυτά επιβεβαιώνονται από τα αποτελέσματα:

[3] Ούτε τώρα είναι πλήρης ο κανόνας. Θα επανέλθουμε...

```
B object created
A object created
C object created
D object created
```

Να γράψουμε λοιπόν για την *DateTime* τον δημιουργό:

```
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
```

Σύμφωνα με τα παραπάνω, όταν θα εκτελείται, αρχικώς θα δημιουργηθεί το υποαντικείμενο της βασικής κλάσης, της *Date*, μετά θα δημιουργηθούν τα μέλη *dtHour*, *dtMin*, *dtSec* και τέλος θα εκτελεσθεί το σώμα όπου θα δώσουμε τιμές στα μέλη (αφού χρειάζονται και έλεγχοι):

```
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Καλά όλα αυτά, αλλά στα μέλη του υποαντικειμένου πώς θα δώσουμε τιμές; Αν προσπαθήσουμε να γράψουμε

```
   dYear = yp;  dMonth = mp;  dDay = dp;
```

μας περιμένει μια δυσάρεστη έκπληξη[4]:

```
Error E2247 dateTime01.cpp 128: 'Date::dYear' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dMonth' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dDay' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
```

Δηλαδή: ο δημιουργός της *DateTime* δεν έχει πρόσβαση στα *dYear*, *dMonth*, *dDay* του υποαντικειμένου *Date*! Για όλα φταίει εκείνο το "`private:`" που έχουμε στη *Date* και στη συνέχεια θα δούμε πώς διορθώνεται.

Εδώ θα δώσουμε άλλη λύση στο πρόβλημά μας, με τη λίστα εκκίνησης:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Τι λέμε εδώ; Δημιούργησε το υποαντικείμενο κλάσης *Date* με παραμέτρους *yp*, *mp*, *dp* και μετά συμπλήρωσε τις τιμές των «νέων» μελών. Με αυτόν τον τρόπο:

[4] Borland C++ v.5.5

- Δεν χρειάζεται να ξαναβάλουμε τους ελέγχους αφού θα τους κάνει ο δημιουργός της *Date*.
- Τα μέλη του υποαντικειμένου *Date* παίρνουν τιμή με τη δημιουργία του. Δεν χρειάζεται να πάρουν τις ερήμην τιμές και μετά να βάλουμε αυτές που πιθανόν έχουν δοθεί.

Βλέπουμε λοιπόν ότι *στη λίστα εκκίνησης μιας παράγωγης κλάσης μπορούμε να καλούμε τον δημιουργό της βασικής για να δημιουργήσει το αντίστοιχο υποαντικείμενο*.

23.3.1 Και Ένας Άλλος Τρόπος

Με βάση αυτά που είπαμε για εκχώρηση τιμής στο υποαντικείμενο της βασικής κλάσης μπορούμε να γράψουμε:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Φυσικά, δεν εννοούμε να προτιμήσεις αυτόν τον τρόπο. Ο πρώτος είναι σαφώς καλύτερος.

23.3.2 Ο Δημιουργός Αντιγραφής

Κατ' αρχάς να ξεκαθαρίσουμε ότι ο *εννοούμενος δημιουργός αντιγραφής της DateTime είναι σωστός*.

Θα γράψουμε όμως εδώ έναν τέτοιο δημιουργό μόνο και μόνο για παράδειγμα. Αλλά, πριν από αυτό θα δώσουμε ένα παράδειγμα της *δυνατότητας υποκατάστασης*. Αν έχουμε ορίσει:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
```

τότε οι

```
   Date d( dt );
   cout << d << endl;
```

θα δώσουν:

```
16.11.2007
```

Δηλαδή: Ένα αντικείμενο της παράγωγης κλάσης μπορεί να χρησιμοποιηθεί ως αρχική τιμή για ένα αντικείμενο της βασικής. Φυσικά, έχουμε τεμαχισμό και στο *d* εκχωρείται η τιμή του υποαντικειμένου κλάσης *Date* του *dt*.

Με βάση αυτό, να πώς μπορούμε να γράψουμε τον δημιουργό αντιγραφής της *DateTime*:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs )
{
   dtHour = rhs.dtHour;  dtMin = rhs.dtMin;  dtSec = rhs.dtSec;
}; // DateTime::DateTime
```

Με το "`Date(rhs)`" καλούμε τον δημιουργό αντιγραφής της *Date* για να δημιουργήσει αντίγράφο του υποαντικειμένου *Date* του *rhs*. Το σώμα του δημιουργού αναλαμβάνει την αντιγραφή των άλλων μελών.[5]

Από τον δημιουργό αντιγραφής παίρνουμε τον τελεστή εκχώρησης όπως μάθαμε στην §21.7, αφού προηγουμένως ορίσουμε την «ασφαλή *swap*». Το βλέπουμε στη συνέχεια.

23.4 Και ο Καταστροφέας ...

Η καταστροφή του αντικειμένου της παράγωγης κλάσης γίνεται με την αντίστροφη σειρά από αυτήν της δημιουργίας:

- Εκτελείται το σώμα του καταστροφέα.
- Καταστρέφονται τα μέλη του αντικειμένου.
- Καλείται (αυτομάτως) ο καταστροφέας της βασικής κλάσης και καταστρέφει το υποαντικείμενο της βασικής κλάσης.

Για παράδειγμα, αν στο πρόγραμμα της §23.3 εφοδιάσεις τις κλάσεις σου με κατάστροφείς του είδους:

```
   ~A() {  cout << "destroying A object" << endl;  }
   ~B() {  cout << "destroying B object" << endl;  }
 . . .
```

θα δεις στο τέλος της εκτέλεσης:

```
destroying D object     (εκτελείται το σώμα του καταστροφέα)
destroying C object     (καταστρέφονται τα μέλη του αντικειμένου)
destroying A object
destroying B object     (καταστρέφεται το υποαντικείμενο της βασικής)
```

Με τον καταστροφέα όμως δεν τελειώσαμε· θα επανέλθουμε...

23.5 Νέες και Παλιές Μέθοδοι

Να δούμε τι άλλο χρειαζόμαστε για τη νέα κλάση εκτός από τον δημιουργό και αυτά που κληρονομούνται από βασική.

Σίγουρα θα χρειαστούμε μεθόδους "*get*" και "*set*" για τα τρία μέλη:

```
   unsigned int getHour() const {  return dtHour;  };
   unsigned int getMin() const {  return dtMin;  };
   unsigned int getSec() const {  return dtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
```

όπου:

```
void DateTime::setHour( int hp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "setHour",
                          DateTimeXptn::hourRange, hp );
   dtHour = hp;
} // DateTime::setHour

void DateTime::setMin( int minp )
{
   if ( minp < 0 || 59 < minp )
```

[5] Αυτά γενικώς. Στην συγκεκριμένη περίπτωση θα ήταν καλύτερο να γράψουμε:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs ), dtHour( rhs.dtHour ), dtMin( rhs.dtMin ), dtSec( rhs.dtSec ) { }
```

```
      throw DateTimeXptn( "setMin",
                          DateTimeXptn::minRange, minp );
   dtMin = minp;
} // DateTime::setMin

void DateTime::setSec( int sp )
{
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "setSec",
                          DateTimeXptn::minRange, sp );
   dtSec = sp;
} // DateTime::setSec
```

Μια μέθοδος που μας ενδιαφέρει είναι η *forward()* –ή ο τελεστής "**+=**"– αλλά τη θέλουμε *να αυξάνει τα δευτερόλεπτα και όχι τις ημέρες*. Γίνεται; Γίνεται:[6]

```
DateTime& DateTime::forward( long int ds )
{
   tm     currD = { dtSec, dtMin, dtHour, getDay(),
                    getMonth()-1, getYear()-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   if ( ds < 0 )
   {
      if ( currT < (-ds) )  // currT + ds < 0
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   else // ds >= 0
   {
      if ( ds > LONG_MAX - currT )  // currT + ds > ULONG_MAX
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   currT += ds;
   currD = *localtime( &currT );
   *(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                       currD.tm_mon+1,
                                       currD.tm_mday );
   dtHour = currD.tm_hour;
   dtMin = currD.tm_min;
   dtSec = currD.tm_sec;
   return *this;
} // Date::forward
```

Πρόσεξε δύο σημεία:

- Χρησιμοποιούμε και εδώ τον "**=**" της βασικής κλάσης (*Date*).
- Πρόσεξε τις επικεφαλίδες της μεθόδου στις δύο κλάσεις:

```
Date& forward( long int dd )
DateTime& forward( long int ds )
```

Διαφέρουν μόνο στον τύπο της επιστρεφόμενης τιμής. Δες όμως το αποτέλεσμα αυτής της ομοιότητας:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
   Date d( dt );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
   d.forward( 10 );
   dt.forward( 10 );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
```

[6] Αν δεν καταλαβαίνεις τις λεπτομέρειες της υλοποίησης ξαναδιάβασε την §22.5.1.4.

```
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
```

Αποτέλεσμα:

```
16.11.2007
16.11.2007 19:30:31
26.11.2007
16.11.2007 19:30:41
```

Αν ορίσουμε:

```
   DateTime& operator+=( long int ds ) {  return forward( ds );  }
```

και αλλάξουμε τίς κλήσεις προς τις *forward()* σε:

```
   d += 10;
   dt += 10;
```

θα πάρουμε τα ίδια αποτελέσματα.

Και στις δύο περιπτώσεις, δράσαμε με την ίδια μέθοδο (τον ίδιο τελεστή) και το ίδιο όρισμα (10) και στα δύο αντικείμενα –*d* κλάσης *Date* και *dt* κλάσης *DateTime*– και η πρώτη προχώρησε κατά 10 ημέρες ενώ η δεύτερη προχώρησε κατά 10 *sec*. Τέτοιες μέθοδοι (τελεστές) λέγονται *πολυμορφικές*. Ο πολυμορφισμός από κληρονομιές βασίζεται στο ότι, για τα αντικείμενα της παράγωγης κλάσης η μέθοδος της παράγωγης κλάσης **υπερισχύει** (overrides) της μεθόδου της βασικής.

♦ ***Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.***

Δηλαδή, δεν υπάρχει περίπτωση να χρησιμοποιήσει ένα αντικείμενο της *DateTime* την *forward()* της *Date*; Ναι, αν ζητηθεί πιο συγκεκριμένα. Στο παραπάνω παράδειγμα δίνουμε:

```
   dt.Date::operator+=( 10 );
```

ή

```
   dt.Date::forward( 10 );
```

και η τιμή του *dt* γίνεται:

```
26.11.2007 19:30:41
```

Με τον πολυμορφισμό δεν τελειώσαμε. Στη συνέχεια θα δούμε μερικές πολύ ενδιαφέρουσες πτυχές του θέματος.

Παρατήρηση:▶

Με το "`this->Date::`" μπορούμε να καλούμε τις μεθόδους του υποαντικειμένου κλάσης *Date* και μέσα στις μεθόδους της παράγωγης κλάσης (*DateTime*). Για παράδειγμα αντί για

```
   *(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                       currD.tm_mon+1,
                                       currD.tm_mday );
```

θα μπορούσαμε να γράψουμε:

```
   this->Date::operator=( Date(currD.tm_year+1900, currD.tm_mon+1,
                               currD.tm_mday) );
```

Με αυτόν τον τρόπο δεν δημιουργείται αντίγραφο του `this` και έχεις ένα (πολύ μικρό) κέρδος.◀

Ας δούμε τώρα ένα άλλο πρόβλημα: Αν κάποιος γράψει `++dt` τότε η τιμή της *dt* θα προχωρήσει κατά μια ημέρα. Αυτό μπορεί να δημιουργήσει σύγχυση. Να την ξαναορίσουμε ώστε να προχωράει κατά 1 *sec*; Ας πούμε ότι αυτό δεν μας ενδιαφέρει· αν θέλουμε να προχωρήσουμε 1 *sec* θα χρησιμοποιούμε τη *forward()*. Τι κάνουμε;

Ορίζουμε στην περιοχή `private` της *DateTime*:

```
   DateTime& operator++() { }
```

Αν τυχόν σου ξεφύγει κάποιο `++dt` μέσα στο πρόγραμμά σου θα το φροντίσει ο μεταγλωττιστής: «`'DateTime& operator++()' is private`.»

Φυσικά, αν κάποτε θελήσεις να χρησιμοποιήσεις τον "`++`" της *Date* μπορείς να δώσεις:

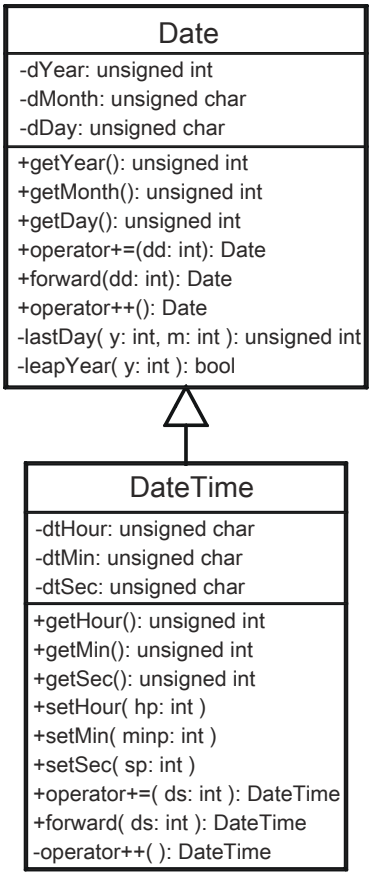
```
dt.Date::operator++();
```

Παρατήρηση: ▶
Αν εσύ επιμένεις να επιφορτώσεις τον "`++`" ώστε η "`++dt`" να αυξάνει τον χρόνο κατά 1 *sec* θα πρέπει, σύμφωνα με αυτά που είπαμε στην §22.2, να την ορίσεις **inline** με βάση τη *forward*:

```
DateTime& operator++()
{  return forward( 1 );  }
```

Αυτός ο ορισμός διαφέρει από τον αντίστοιχο της *Date* (§22.5.1.6) μόνον στον τύπο του αποτελέσματος.◀

Στο Σχ. 23-3 βλέπεις ένα λεπτομερέστερο διάγραμμα κληρονομιάς.



Σχ. 23-3 Η κλάση *DateTime* είναι παράγωγη της κλάσης *Date*. Κληρονομεί από αυτήν τα *dYear*, *dMonth*, *dDay*, *getYear*, *getMonth*, *getDay*. Δεν κληρονομεί τις `operator+=`, *forward*, `operator++` (έχει δικές της).

23.5.1 Ο Τελεστής Εκχώρησης

Αν χρειάζεται να ορίσεις τον τελεστή εκχώρησης για την παράγωγη κλάση όρισέ τον με βάση τον δημιουργό αντιγραφής που, σύμφωνα με τον «κανόνα των τριών», μάλλον θα πρέπει να έχεις ορίσει. Χρησιμοποιώντας το πάγιο «πατρόν», δεν έχεις παρά να ορίσεις τη *swap()*.

Ορίζουμε τη *swap()* χρησιμοποιώντας τη *swap()* της βασικής. Στο παράδειγμά μας:

```
void DateTime::swap( DateTime& rhs )
{
   this->Date::swap( rhs );
   std::swap( dtHour, rhs.dtHour );
   std::swap( dtMin, rhs.dtMin );
   std::swap( dtSec, rhs.dtSec );
} // DateTime::swap
```

Η "`this->Date::swap(rhs)`" (ή, επί το απλούστερο: "`Date::swap(rhs)`") ασχολείται μόνο με το υποαντικείμενο *Date* του *rhs*.[7]

Μετά ο δρόμος είναι γνωστός:

```
DateTime& DateTime::operator=( const DateTime& rhs )
{
   if ( &rhs != this )
   {
      DateTime tmp( rhs );
      swap( tmp );
   }
   return *this;
} // DateTime::operator=
```

[7] Μα δεν έχουμε ορίσει *swap()* για τη *Date*! Ας την ορίσουμε:

```
void Date::swap( Date& rhs )
{
   std::swap( dYear, rhs.dYear );
   std::swap( dMonth, rhs.dMonth );
   std::swap( dDay, rhs.dDay );
} // Date::swap
```

23.6 Καθολικές Συναρτήσεις

Όπως στις μεθόδους, έτσι και στην υλοποίηση καθολικών συναρτήσεων για την παράγωγη κλάση μπορούμε να χρησιμοποιήσουμε συναρτήσεις που γράψαμε για τη βασική κλάση. Ας δούμε δύο παραδείγματα.

Επιφορτώνουμε τον "==" για τη *DateTime* ως εξής:

```
bool operator==( const DateTime& lhs, const DateTime& rhs )
{
   return ( static_cast<Date>(lhs) == static_cast<Date>(rhs) &&
            lhs.getHour() == rhs.getHour() &&
            lhs.getMin() == rhs.getMin() &&
            lhs.getSec() == rhs.getSec() );
} // operator==( const DateTime
```

και τον "<<":

```
ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return tout << static_cast<Date>( rhs ) << ' '
               << rhs.getHour() << ':' << rhs.getMin() << ':'
               << rhs.getSec();
} // operator<<( ostream& tout, const DateTime
```

23.7 Κλάση Εξαιρέσεων Παράγωγης Κλάσης

Θα τηρούμε τον εξής κανόνα:

- ***Η κλάση εξαιρέσεων της παράγωγης κλάσης είναι παράγωγη της κλάσης εξαιρέσεων της βασικής κλάσης.***

Θα έχουμε, για παράδειγμα:

```
struct DateTimeXptn :  public DateXptn
{
   enum { hourRange = 50, minRange, secRange  };
   DateTime errDateTimeVal;
   DateTimeXptn( const char* mn, int ec,
                 int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : DateXptn( mn, ec, ev1, ev2, ev3 ) {  }
   DateTimeXptn( const char* mn, int ec, int ev1, const DateTime& edt )
      : DateXptn( mn, ec, ev1, edt )
   {  errDateTimeVal = edt;  }
}; // DateTimeXptn
```

Πρόσεξε:

- Το "`hourRange = 50`": με αυτόν τον τρόπο επιδιώκουμε να διαχωρίσουμε τις περιοχές κωδικών σφάλματος των *DateXptn* (0 .. *DateXptn*::*cannotWrite*) και *DateTimeXptn* (50.. *DateTimeXptn*::*secRange*).
- Τη σύλληψη των εξαιρέσεων· αν κάπου πιάνεις εξαιρέσεις και των δύο τύπων θα πρέπει να βάλεις:

  ```
  catch( DateTimeXptn& x )
  { . . . }
  catch( DateXptn& x )
  { . . . }
  ```

  Αν βάλεις τις **catch** με την αντίστροφη σειρά η "`catch(DateXptn& x)`" δεν θα αφήνει να περάσει οποιαδήποτε εξαίρεση στην "`catch(DateTimeXptn& x)`", αφού κάθε `DateTimeXptn` είναι και `DateXptn`.

Και μια παρατήρηση για το συγκεκριμένο παράδειγμα: Ένα αντικείμενο κλάσης *DateTimeXptn* έχει μέλος κλάσης *DateTime* αλλά έχει κληρονομήσει και μέλος κλάσης *Date* (*errDateVal*). Στο κληρονομημένο μέλος δίνουμε τιμή με τη λίστα εκκίνησης του δεύτερου

δημιουργού, από το υποαντικείμενο κλάσης *Date* του *edt*, με τον σχετικό τεμαχισμό. Ολόκληρο το *edt* αντιγράφεται στο νέο μέλος *errDateTimeVal*.

23.8 "private" ή "protected"

Λέγαμε στην §23.3 ότι «*ο δημιουργός της DateTime δεν έχει πρόσβαση στα dYear, dMonth, dDay του υποαντικειμένου Date! Για όλα φταίει εκείνο το "*`private`*:" που έχουμε στη Date*». Αυτό βέβαια ισχύει και για οποιαδήποτε μέθοδο της *DateTime*.

Παρ' ολα αυτά, όπως φάνηκε από αυτά που είδαμε στις προηγούμενες παραγράφους:

- Μπορούμε να αλλάζουμε τις τιμές των μελών του υποαντικειμένου της βασικής με κάποιον δημιουργό ή άλλες μεθόδους (π.χ. *swap()*) και –αν χρειαστεί– με κάποια "*set*".
- Μπορούμε να παίρνουμε τις τιμές των μελών του υποαντικειμένου της βασικής με τις "*get*".

Σε πολλές περιπτώσεις πάντως, όταν έχουμε πολύπλοκους υπολογισμούς, τα παραπάνω δεν είναι ικανοποιητικά. Εκτός από αυτό, συχνά στις περιοχές `private` υπάρχουν «κρυφές» μέθοδοι ή βοηθητικές συναρτήσεις που μας είναι χρήσιμες για την υλοποίηση των μεθόδων της παράγωγης κλάσης.

Η C++ μας δίνει λύση στα προβλήματα αυτά με μια τρίτη κατηγορία δηλώσεων, τις δηλώσεις "`protected`".

♦ ***Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της.***

Σημείωση: ▶

Δες μια άλλη κάπως ακριβέστερη διατύπωση: *Ό,τι υπάρχει στην περιοχή "`protected`" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης*. Διάβασε την επόμενη υποπαράγραφο.◀

Ας δούμε ένα παράδειγμα· αλλάζουμε τη *Date* ως εξής:

```
class Date
{
// . . .
public:
// . . .
protected:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   static unsigned int lastDay( int y, int m );
private:
   static bool leapYear( int y );
}; // Date
```

Τώρα στον δημιουργό της *DateTime* μπορείς να γράψεις, αφού βάλεις τους ελέγχους που ξέρουμε:

```
   if ( yp <= 0 ) throw . . .
   dYear = yp;
   if ( mp <= 0 || 12 < mp ) throw . . .
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
```

αντί για

```
   :  Date( yp, mp, dp )
```

ή

```
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
```

Αλλά προσοχή: αυτό δεν σημαίνει ότι πρέπει να γράψεις έτσι τον δημιουργό. Όπως βλέπεις, ξαναγράφουμε και εδώ τους ελέγχους που έχουμε γράψει στον δημιουργό και στις "*set*" της *Date*. Οι δύο τρόποι, που είδαμε πιο πριν είναι προτιμότεροι διότι βασίζονται στον δημιουργό της *Date* που έχει τους ελέγχους.

Αν πάμε όμως στη *forword()* τα πράγματα αλλάζουν: οι

```
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
```

είναι προτιμότερες από την

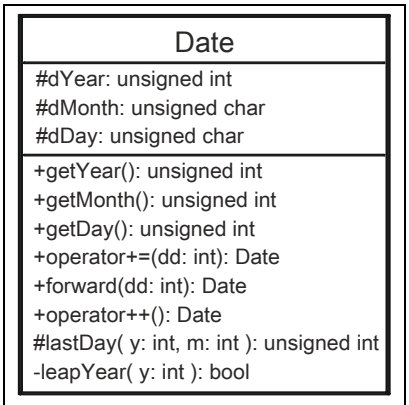
```
   *(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                       currD.tm_mon+1,
                                       currD.tm_mday );
```

διότι ο δημιουργός που καλείται θα κάνει ελέγχους που είναι άχρηστοι.

Όπως λέγαμε και στην §20.8.1, στα διαγράμματα της UML, ό,τι δηλώνεται σε περιοχή "`protected`" σημειώνεται με "`#`". Στο Σχ. 23-4 βλέπεις διαγραμματική παράσταση της *Date*, όπως την αλλάξαμε παραπάνω.

Πολλοί πεπειραμένοι προγραμματιστές βάζουν πάντοτε τα μέλη σε περιοχή "`private`" ενώ σε περιοχή "`protected`" μπορεί να βάλουν βοηθητικές συναρτήσεις ή μεθόδους.

Άλλοι, χωρίς συζήτηση, αντικαθιστούν το "`private`" με το "`protected`" σε κάθε κλάση που πρόκειται να κληρονομηθεί. Αν θέλεις να κάνεις και εσύ το ίδιο πάρε υπόψη σου τα εξής:



Σχ. 23-4 Η κλάση *Date* με τα μέλη *dYear*, *dMonth*, *dDay* και τη βοηθητική συνάρτηση *lastDay* σε περιοχή `protected`.

- ***Αν χρησιμοποιείς την αλλαγή περιορισμού πρόσβασης από "`private`" σε "`protected`" –για να μπορείς να αλλάζεις τις τιμές των μελών του υποαντικειμένου της βασικής κλάσης– θα πρέπει να βάζεις και τους απαραίτητους ελέγχους κατά περίπτωση.***

Φυσικά, με αυτόν τον τρόπο, πληθαίνουν τα σημεία των ελέγχων πράγμα που κάνει πιο πολύπλοκη τη κάθε απόπειρα ενημέρωσης/τροποποίησης της βασικής κλάσης. Για τον λόγο αυτόν, ακόμη και με τη αλλαγή σε "`protected`", καλό είναι

- ***Να προσπαθείς να ελαχιστοποιείς την κατ' ευθείαν πρόσβαση στα μέλη του υποαντικειμένου γράφοντας τις μεθόδους της παράγωγης κλάσης με χρήση των μεθόδων και των δημιουργών της βασικής.***

23.8.1 * "`protected`": Ψιλά Γράμματα

Επανερχόμαστε τώρα στον κανόνα-ορισμό της προηγούμενης παραγράφου: «*Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της*» και στη διευκρινιστική σημείωση «*Ό,τι υπάρχει στην περιοχή "`protected`" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης.*» Ας πούμε ότι έχουμε:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   void display( ostream& tout ) const {  tout << mb;  }
protected:
   int mb;
}; // B
```

```
class D2: public B
{
public:
   D2( int bp=2, int d2p=0 ): B(bp)  {  md2 = d2p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md2;  }
private:
   int md2;
}; // D2
```

Θέλουμε να εφοδιάσουμε τη *D2* με μια μέθοδο:

```
   void xchangeB( B& b );
```

που θα ανταλλάσσει τις τιμές των μελών *mb* του αντικειμένου και του *b*. Αν

```
   B b;
   D2 d2a( 2, 11 );
```

τότε οι:

```
   d2a.xchangeB( b );
   d2a.display( cout );  cout << endl;
   b.display( cout );  cout << endl;
```

θα πρέπει να δώσουν:

```
0 11
2
```

Θέλουμε να υλοποιήσουμε τη μέθοδο ως εξής:

```
   void xchangeB( B& b )  {  std::swap( b.mb, mb );  }
```

αλλά ο μεταγλωττιστής[8] έχει αντιρρήσεις: «`Error E2247 testDerived.cpp 31: 'B::mb' is not accessible in function D2::xchangeB(B &)`».

Εδώ τι έχουμε;

- Στη βασική κλάση (*B*) έχουμε δηλώσει "`protected: int mb`".
- Στην παράγωγη κλάση (*D1*), μια μέθοδος προσπαθεί να διαχειρισθεί το μέλος *mb* ενός αντικειμένου *b* κλάσης *B*.

Ο μεταγλωττιστής μας λέει ότι το "`B::mb`" δεν είναι προσβάσιμο από τη συνάρτηση-μέλος "`D2::xchangeB(B &)`".

Αν βάλουμε:

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

ο μεταγλωττιστής δεν έχει πρόβλημα με τη μέθοδο. Αν έχουμε δηλώσει

```
   D2 d2a( 2, 11 ), d2b( 3, 13 );
```

οι

```
   d2a.xchangeB( d2b );
   d2a.display( cout );  cout << endl;
   d2b.display( cout );  cout << endl;
```

θα δώσουν:

```
3 11
2 13
```

Τώρα όμως δεν είναι δεκτή η "`d2a.xchangeB( b )`".

Αν ορίσουμε

```
class DD2: public D2
{
public:
   DD2( int bp=1, int d2p=0, int dd2p=0 )
      : D2(bp, d2p)  {  mdd2 = dd2p;  }
   void display( ostream& tout ) const
   {  D2::display( tout );  tout << ' ' << mdd2;  }
```

[8] Borland BC++, v.5.5.

```
private:
   int mdd2;
}; // DD2
```

και δηλώσουμε:

```
   DD2 dd2( 5, 17, 23 );
```

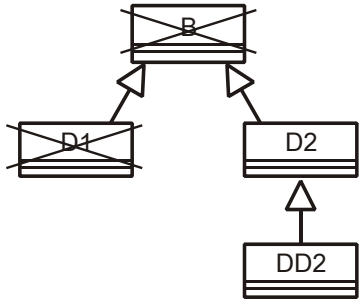
η

```
   d2a.xchangeB( dd2 );
```

είναι δεκτή και δουλεύει.

Αν όμως ορίσεις

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 )
      : B(bp)  {  md1 = d1p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
private:
   int md1;
}; // D1
```



Σχ. 23-5 Οι μέθοδοι της *D2* «βλέπουν» τα "`protected`" μέλη των υποαντικειμένων *B* των αντικειμενων της *D2* και της *DD2*· όχι της *B* και της *D1*.

Αλλάζουμε τώρα την *xchangeB()* σε

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

Και πάλι το ίδιο πρόβλημα: δεν επιτρέπεται πρόσβαση προς το "`b.mb`"!

Διατυπώνουμε λοιπόν τον κανόνα μας ως εξής (Σχ. 23-5):

- ***Αν έχουμε μια (βασική) κλάση B στην οποίαν υπάρχει περιοχή "`protected`", μια κλάση D, παράγωγη (αμέσως ή εμμέσως) της B και μια μέθοδο f της D, η f έχει πρόσβαση στα μέλη των περιοχών "`protected`" των υποαντικειμένων κλάσης B των αντικειμένων κλάσης D και των παραγώγων της D.***

23.9 Εικονικές Μέθοδοι

Στην §23.5 λέγανε: «*Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.*» Είναι όμως πάντα έτσι; Δες ένα πρόβλημα: Δηλώνουμε

```
   Date* pind[2];
```

και παίρνουμε μνήμη:

```
   pind[0] = new Date( 2002, 2, 2 );
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

Το δεύτερο είναι σωστό; Είναι και παραείναι! Αφού «*Οπουδήποτε … μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D*» όλα είναι εντάξει. Για δες όμως τη συνέχεια:

```
   pind[0]->forward( 10 );
   pind[1]->forward( 10 );
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
   ++(*pind[0]);
   ++(*pind[1]);
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
```

Αποτέλεσμα:

```
12
13
13
14
```

Δηλαδή; Και για το δεύτερο αντικείμενο χρησιμοποίησε τη *forward()* και τον "++" της βασικής. Η ζημιά έγινε από τον μεταγλωττιστή· όταν έκανε τη μεταγλώττιση σημείωσε ότι τα βέλη του pind δείχνουν αντικείμενα κλάσης *Date* και αυτό ήταν... Όταν έρχεται η ώρα του υπολογισμού χρησιμοποιεί τις μεθόδους της *Date*.

Τώρα σου έρχεται αγανάκτηση: «Ε, δεν θα χρησιμοποιούμε βέλη και τελειώσαμε!» Εύκολα το λες, δύσκολα το κάνεις. Ας πούμε ότι έχουμε:

```
   DateTime dt( 2003, 3, 3, 3, 33, 33 );
```

και τη συνάρτηση:

```
void plus10( Date& d )
{
   d.forward( 10 );
} // plus10
```

Οι εντολές

```
   cout << dt << endl;
   plus10( dt );
   cout << dt << endl;
```

θα δώσουν:

```
3.3.2003 3:33:33
13.3.2003 3:33:33
```

Δηλαδή, κλήθηκε η *forward()* της *Date* και όχι της *DateTime*. Γιατί; Αν το έχεις ξεχάσει, να υπενθυμίσουμε ότι

- οι παράμετροι αναφοράς είναι «μεταμφιεσμένα» βέλη και ότι
- όταν θέλουμε να περάσουμε σε συνάρτηση αντικείμενο θα χρησιμοποιήσουμε παράμετρο αναφοράς (με `const` ή χωρίς), διότι τα αντικείμενα είναι συνήθως μεγάλα.

Καταλαβαίνεις λοιπόν ότι δεν μπορείς να πεις τόσο εύκολα ότι δεν θα χρησιμοποιήσεις βέλη.

Πώς διορθώνεται αυτό; Ως εξής:

♦ ***Όποιες μεθόδους ξαναορίζουμε στην παράγωγη κλάση τις δηλώνουμε στη βασική κλάση ως "`virtual`" (εικονικές).***

```
class Date
{
// . . .
public:
   virtual const Date& operator+=( long int dd );
   virtual const Date& forward( long int dd );
   virtual Date& operator++();
// . . .
}; // Date
```

Μετά από αυτή τη διόρθωση οι εντολές του πρώτου παραδείγματος θα δώσουν:

```
12
3
13
3
```

ενώ από το δεύτερο θα πάρουμε:

```
3.3.2003 3:33:33
3.3.2003 3:33:43
```

Τι κάνει αυτή η «μαγική» λέξη;

- Ο χαρακτηρισμός "`virtual`" (εικονικός) μπαίνει σε μεθόδους της *βασικής* κλάσης όταν στην παράγωγη κλάση υπάρχει μέθοδος με το ίδιο όνομα (πολυμορφική). Στην περίπτωσή μας βάλαμε "`virtual`" τις `operator+=()`, *forward()*, `operator++()` της *Date* διότι αυτές ορίζονται ξανά και στην παράγωγή της *DateTime*.

- Το αποτέλεσμά του είναι το εξής: προειδοποιεί τον μεταγλωττιστή να μην βιαστεί να «δέσει» τη συγκεκριμένη μέθοδο με τα βέλη προς αντικείμενα της βασικής αλλά να αναβάλει την απόφασή του μέχρι τη στιγμή της εκτέλεσης της εντολής ώστε να μπορεί να «δει» *την κλάση του αντικειμένου που δείχνει το βέλος εκείνη τη στιγμή*.

Να βάλουμε "`virtual`" και στις αντίστοιχες μεθόδους της παράγωγης κλάσης; Πρόσεξε: Αν κατάλαβες τι ακριβώς συμβαίνει με το "`virtual`" θα έχεις καταλάβει ότι και αυτές είναι "`virtual`" είτε βάλουμε είτε δεν βάλουμε.[9] Καλύτερα λοιπόν να γράψουμε το "`virtual`" και στην παράγωγη κλάση για λόγους τεκμηρίωσης. Γιατί να μην δούμε την αλήθεια κατάματα;...

23.10 * Υλοποίηση Εικονικών Συναρτήσεων

Θα πούμε τώρα δυο λόγια για την υλοποίηση των εικονικών μεθόδων, παρ' όλο που αυτό είναι έξω από το αντικείμενο αυτού του βιβλίου.

Με τις κλάσεις του παραδείγματος της §23.3 (κάπως τροποποιημένες)

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual void display( ostream& tout ) const {  tout << mb;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B

class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
   virtual void g( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

κάνουμε το εξής πείραμα: Δηλώνουμε δύο αντικείμενα:

```
   B   bo;
   D1  d1o;
```

και ζητούμε:

```
   cout << "sizeof_bo: " << sizeof(bo) << endl;
   cout << "sizeof_d1o: " << sizeof(d1o) << endl;
```

Αποτέλεσμα:[10]

```
sizeof_bo: 8
sizeof_d1o: 12
```

Αλλά σε ένα αντικείμενο κλάσης *B* έχουμε ένα μέλος τύπου `int` ενώ τα αντικείμενα κλάσης *D1* έχουν δύο τέτοια μέλη. Ο μεταγλωττιστής που χρησιμοποιούμε παριστάνει τις τιμές τύπου `int` με 4 ψηφιολέξεις. Και στις δύο περιπτώσεις «περισσεύουν» 4 ψηφιολέξεις. Με αυτά υλοποιείται μια «κρυφή» μεταβλητή-βέλος, ας την πούμε *vfptr*, που δείχνει έναν πίνακα με τις εικονικές μεθόδους (για την ακρίβεια τις διευθύνσεις των εικονικών μεθό-

[9] «*Once virtual, always virtual!*»

[10] g++ (Dev-C++).

δων) που χρησιμοποιούν τα αντικείμενα της κλάσης και ονομάζεται **πίνακας εικονικών μεθόδων / συναρτήσεων** (virtual method / function table) ή απλώς **vtable**.

Κατά τη διάρκεια της μεταγλώττισης:

- Στον vtable της *B* θα εισαχθούν οι *B*::*f* και *B*::*display*.
- Στον vtable της *D1* θα εισαχθούν οι *B*::*f* (που κληρονομείται από τη *B*), *D1*::*display* και *D1*::*g*.
- Κάθε αντικείμενο κλάσης *B*, όπως το *bo*, εφοδιάζεται με το «κρυφό» βέλος *vfptr* και στους δημιουργούς της κλάσης μπαίνουν εντολές που το βάζουν να δείχνει τον vtable της *B*.
- Τα αντίστοιχα γίνονται και με τα αντικείμενα κλάσης *D1*.

Κατά τη διάρκεια της εκτέλεσης αν ζητηθεί η εκτέλεση κάποιας εικονικής μεθόδου ενός αντικειμένου αναζητείται η διεύθυνσή της στον πίνακα που δείχνει το *vfptr* και εκτελείται η σωστή μέθοδος.

23.10.1 "`virtual`" και Τεμαχισμός

Είναι δυνατόν να βάλουμε το "`virtual`" και να μην έχουμε τη σωστή συμπεριφορά του αντικειμένου; Και βέβαια, αν δεν το αφήνουμε να λειτουργήσει. Δες ένα

Παράδειγμα ✎

Χρειαζόμαστε μια συνάρτηση, ας την πούμε *plus10Local()*, που θα τροφοδοτείται με ένα αντικείμενο *Date* ή *DateTime* –όπως η *plus10()*– και θα προχωρεί την τιμή του κατά 10 ημέρες ή *sec* αντιστοίχως αλλά μόνο για τις επεξεργασίες που γίνονται μέσα στη συνάρτηση· η αλλαγή δεν θα βγαίνει προς τη συνάρτηση που θα καλεί την *plus10Local()*.

Θα έλεγε κανείς –με αρκετή επιπολαιότητα– ότι εδώ μας δίνεται η ευκαιρία να «γλυτώσουμε» από τα βέλη και να γράψουμε:

```
void plus10Local( Date d )
{
   d.forward( 10 );
// . . .
} // plus10Local
```

Έτσι κάναμε ένα σοβαρό λάθος: όταν εκτελείται η "`plus10Local(dt)`" καλείται ο δημιουργός αντιγραφής της *Date* για να αντιγράψει το *dt* στο *d*. Το *dt*; Όχι! Το υποαντικείμενο κλάσης *Date* του *dt*. Έχουμε δηλαδή *τεμαχισμό* και το "`virtual`" δεν έχει ευκαιρία να δουλέψει...

✎✎✎

Δυστυχώς, το λάθος αυτό γίνεται με τρόπο που είναι τελείως «νόμιμος» για τον μεταγλωττιστή και έτσι δεν θα μας δώσει κάποιο μήνυμα.

Καταλαβαίνεις τώρα ότι η συνήθεια να περνούμε τα αντικείμενα με παραμέτρους αναφοράς δεν έχει να κάνει μόνο με την αποφυγή μιας αντιγραφής.[11]

Παρατήρηση: ▶

Μια και συζητούμε για αντιγραφές, δες ένα πρόβλημα που προκύπτει: Έστω ότι βάζουμε παράμετρο "`const Date& d`". Τώρα η "`d.forward(10)`" είναι παράνομη λόγω του "`const`". Για να κάνουμε τη δουλειά μας θα πρέπει να αντιγράψουμε το *d* σε ένα αντικείμενο, τοπικό στην *plus10Local*. Αλλά, τι τύπου θα είναι αυτό το τοπικό αντικείμενο; Σε λίγο θα το μάθουμε και αυτό...◀

[11] Ο κανόνας OBJ33 του (CERT 2009) λέει: *Μην τεμαχίζεις πολυμορφικά αντικείμενα* (*Do not slice polymorphic objects*).

23.10.2 Κάποια Σχόλια...

- Ο όρος **virtual** αποδόθηκε στα ελληνικά ως «*εικονική*»· θα βρεις επίσης και το όρο «*νοητή*». Και οι δύο μεταφράσεις είναι εξ ίσου παραπλανητικές. Όπως είδαμε μέχρι τώρα, οι μέθοδοι που δηλώνονται ως "`virtual`" είναι πραγματικότατες. Μια (περιφραστική) μετάφραση «*δυναμικώς επιλεγόμενη*» θα ήταν πιο κοντά στην πραγματικότητα. Πάντως στη συνέχεια θα δούμε μεθόδους που είναι γνησίως εικονικές.
- Γιατί να μην είναι όλες οι μέθοδοι "**virtual**" να τελειώνουμε; Διότι ο μηχανισμός επιλογής της σωστής μεθόδου χρειάζεται χρόνο εκτέλεσης. Γιατί λοιπόν να φορτώνουμε το πρόγραμμά μας με άχρηστες δουλειές για μεθόδους που δεν πρόκειται να αλλάξουμε, π.χ. για τις "*get*" και "*set*";
- Το πρόβλημα που λύνεται με τις εικονικές μεθόδους υπάρχει και στις άλλες αντικειμενοστρεφείς γλώσσες και οι λύση είναι παρόμοια: «**καθυστερημένη**» **επιλογή** (**πρόσδεση**) (late ή deferred binding) μεθόδου. Οι «συγγενεις» γλώσσες –Java και C#– μπορεί να μην χρησιμοποιούν (φανερά) βέλη αλλά χρησιμοποιούν «κρυμμένα» (αναφορές). Έτσι
  - Στη Java κάθε μέθοδος είναι κατ' αρχήν "**virtual**" και ο προγραμματιστής θα πρέπει *να δηλώσει ποιες δεν θέλει* να είναι.
  - Η C# ακολουθεί τη C++: κάθε μέθοδος, κατ' αρχήν, *δεν είναι* "**virtual**" και ο προγραμματιστής θα πρέπει να δηλώσει ποιες θέλει να είναι.

23.11 Εικονικός Καταστροφέας

Επιστρέφουμε στο παράδειγμα της §23.3, όπως το εμπλουτίσαμε με τους καταστροφείς στην §23.4. Δίνουμε τις εντολές:

```
B* pd( new D );
delete pd;
```

και παίρνουμε:

```
B object created
A object created
C object created
D object created
destroying B object
```

Το ενδιαφέρον βρίσκεται στην τελευταία γραμμή, «`destroying B object`», που δείχνει ότι καλείται ο καταστροφέας της βασικής κλάσης για να καταστρέψει ένα αντικείμενο της παράγωγης κλάσης, το `*`*pd*. Πώς έγινε αυτό; Η απόφαση για το είδος του καταστροφέα πάρθηκε πολύ νωρίς, στη μεταγλωττιση.

Πώς διορθώνεται αυτό; Όπως είδαμε στην προηγούμενη παράγραφο:

♦ ***Ο καταστροφέας της βασικής κλάσης πρέπει να δηλώνεται "`virtual`".***

Πράγματι, αν ορίσουμε:

```
virtual ~B() {  cout << "destroying B object" << endl;  }
```

θα πάρουμε:

```
B object created
A object created
C object created
D object created
destroying D object
destroying C object
destroying A object
destroying B object
```

Στην §21.5 λέγαμε «*Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα "`~K() {}`" στις κλάσεις που δεν χρειάζονται. Αργότερα θα δούμε ότι σε*

μερικές περιπτώσεις αυτό είναι απαραίτητο.» Τώρα βλέπεις ότι αυτό είναι απαραίτητο στις κλάσεις που πρόκειται να κληρονομηθούν· και μάλιστα θα πρέπει να ορίζεις:

```
   virtual ~K() { };
```

Παράδειγμα

Θα δούμε στη συνέχεια ότι η C++ έχει μια βασική κλάση εξαιρέσεων, την `std::exception`, από την οποία παράγονται όλες οι κλάσεις εξαιρέσεων που χρησιμοποιούν οι συναρτήσεις της βιβλιοθήκης της. Αυτή η κλάση είναι περίπου έτσι:

```
struct exception
{
   exception() throw() { }
   virtual ~exception() throw();
   virtual const char* what() const throw();
}; // exception
```

Όπως βλέπεις, ο καταστροφέας δεν ορίζεται αλλά έχει στη δήλωσή του το "`virtual`".

Έτσι, μπορούμε να βάλουμε στις δικές μας κλάσεις:

```
struct DateXptn :  public exception
{
// . . .
   virtual ~DateXptn() throw() { };
}; // DateXptn
```

και

```
struct DateTimeXptn :  public DateXptn
{
// . . .
   virtual ~DateTimeXptn() throw() { };
}; // DateTimeXptn
```

Αν θελήσεις να δοκιμάσεις τα παραπάνω θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <exception>`".

23.12 Περί Πολυμορφισμού

Αφού είδαμε διάφορες πλευρές –τεχνικές κυρίως– του πολυμορφισμού ας ολοκληρώσουμε –προς το παρόν– αυτό το θέμα, κυρίως για να ξεδιαλύνουμε μερικές παρεξηγήσεις της μορφής «πολυμορφισμός είναι επιφόρτωση ή υπερίσχυση ή μέθοδοι **virtual**».

Γενικώς, μπορούμε να ορίσουμε τον **πολυμορφισμό** (polymorphism) ως τη δυνατότητα να χειριζόμαστε (εν μέρει συνήθως) έναν τύπο *A* όπως τον τύπο *B*.

Στα πλαίσια του αντικειμενοστρεφούς προγραμματισμού αυτό εξειδικεύεται ως: η δυνατότητα αντικειμένων *διαφορετικών κλάσεων* να αποκρίνονται στο ίδιο μηνυμα με αποκρίσεις –όχι ταυτόσημες, αλλά– που εξαρτώνται από την κλάση του κάθε αντικειμένου.

Αυτά που μάθαμε μέχρι τώρα αφορούν την υλοποίηση στη C++ του λεγόμενου **πολυμορφισμού υποκλάσεων** (subclass ή subtype). Δηλαδή: αντικείμενα διαφορετικών τύπων που ανήκουν στην ίδια ιεραρχία κλάσεων μπορεί να δεχθούν κλήση μεθόδου ή τελεστή με το ίδιο όνομα (π.χ. `d.forward(10)` ή `d += 10`) και αποκρίνονται με τρόπο που εξαρτάται από την κλάση του αντικειμένου (π.χ. αν το *d* είναι κλάσης *Date* προχωρεί την ημερομηνία κατά 10 ημέρες ενώ αν είναι κλάσης *DateTime* προχωρεί τον χρόνο κατά 10 *sec*).

Πώς γίνονται αυτά; Η παράγωγη κλάση (υποκλάση) κληρονομεί από τη βασική (υπερκλάση) τη μέθοδο (τελεστή) που μας ενδιαφέρει (π.χ.: `const Date& forward(long int dd)`). Στην παράγωγη κλάση *επιφορτώνουμε*(;) μια μέθοδο με το ίδιο όνομα και *παρόμοια επικεφαλίδα* (π.χ.: `const DateTime& forward( long int dd)`). Η μέθοδος αυτή

υπερισχύει[12] αυτής που κληρονομήθηκε και *σχεδόν* κάθε φορά που στέλνουμε το σχετικό μήνυμα σε αντικείμενο της παράγωγης αυτό αποκρίνεται με την υπερισχύουσα μέθοδο. Για να γίνει εκείνο το «*σχεδόν* κάθε φορά» απλώς «κάθε φορά» δηλώνουμε τις μεθόδους μας "`virtual`".

Στην παραπάνω ιστορία έχουμε αφήσει κάποια πράγματα που χρειάζονται διευκρίνιση: ένα ερωτηματικό(;) και μια «*παρόμοια επικεφαλίδα*».

- Πρώτα το «*επιφορτώνουμε*(;)». ΔΕΝ επιφορτώνουμε, τουλάχιστον με την έννοια που ξέρουμε μέχρι τώρα. Στην επιφόρτωση συναρτήσεων μάθαμε ότι οι διάφορες συναρτήσεις με το ίδιο όνομα θα πρέπει να έχουν *διαφορετικές υπογραφές*. Δηλαδή θα πρέπει να έχουν *διαφορά* στις παραμέτρους (ώστε να μπορεί ο μεταγλωττιστής να επιλέξει). Οι πολυμορφικές μέθοδοι *πρέπει να έχουν τις ίδιες ακριβώς παραμέτρους*.
- Πόσο «*παρόμοια επικεφαλίδα*»; Αφού έχουμε ίδιο όνομα και ίδιες παραμέτρους, το «*παρόμοια*» πάει στον τύπο (επιστρεφόμενης τιμής) της συνάρτησης. Ο τύπος μπορεί να είναι ο ίδιος ή –αν δεν είναι– θα πρέπει να είναι *μετατρέψιμος* τύπος βέλους ή αναφοράς. Όπως θα δεις παρακάτω, ο τύπος βέλους της βασικής κλάσης (π.χ. `Date*` ή `Date&`) μπορεί να μετατραπεί σε τύπο βέλους προς την παράγωγη (π.χ. `DateTime*` ή `DateTime&`)· ο τύπος της μεθόδου της παράγωγης κλάσης είναι βέλος παράγωγης κλάσης.[13]

Για παράδειγμα, μπορείς να δεις αυτά τα πράγματα στη μέθοδο *forward()* ή στον "`+=`" των *Date* και *DateTime*:

```
const Date& Date::forward( long int dd )
const DateTime& DateTime::forward( long int ds )
```

ή

```
const Date& Date::operator+=( long int dd )
const DateTime& DateTime::operator+=( long int ds )
```

Και στις δύο περιπτώσεις μπορούσαμε να τις κάνουμε "`virtual`" και τις κάναμε.

Αυτός ο –δυναμικός– πολυμορφισμός ονομάζεται **πολυμορφισμός χρόνου εκτέλεσης** (run-time).

Παρατήρηση: ▶

Αν στη βασική κλάση, μια μέθοδος "`virtual`" έχει προδιαγραφή εξαιρέσεων, οι υπερισχύουσες μέθοδοι στις παράγωγες κλάσεις θα πρέπει να έχουν την ίδια ακριβώς προδιαγραφή εξαιρέσεων. Στο παράδειγμα της §23.11 μας δίνεται η βασική κλάση με υπογραφή καταστροφέα:

```
virtual ~exception() throw();
```

Στην παράγωγη κλάση (*DateXptn*) θα πρέπει να βάλουμε:

```
virtual ~DateTimeXptn() throw() { };
```

Αν παραλείψεις το "`throw()`" ο μεταγλωττιστής θα σου βγάλει λάθος.◀

23.12.1 Πολυμορφισμός Χρόνου Μεταγλώττισης(;)

Ας πούμε ότι μας ενδιαφέρει να κάνουμε πολυμορφικό τον μεταθεματικό "++". Όπως έχουμε ήδη πει, αυτός *δεν μπορεί να επιστρέψει βέλος ή αναφορά* και θα είναι:

```
Date Date::operator++( int )
DateTime DateTime::operator++( int )
```

Μπορούμε να το αφήσουμε έτσι· στην παράγωγη θα έχουμε απόκρυψη του τελεστή της βασικής. Αλλά

---

[12] Από την στιγμή που έχουμε *υπερίσχυση* παύουμε να μιλούμε για *επιφόρτωση*. Κρατούμε αυτόν τον όρο για την περίπτωση που η επικεφαλίδα δεν είναι *παρόμοια*. Το συζητούμε στη συνέχεια.

[13] Αυτό ονομάζεται *συμμεταβλητότητα* (covariance).

- Θα έχουμε διαφορετική συμπεριφορά των αντικειμένων της παράγωγης όταν τα χειριζόμαστε με βέλη και αναφορές της βασικής.
- Δεν μπορούμε να βάλουμε το "`virtual`" για να λύσουμε το πρόβλημα.

Αυτός ο πολυμορφισμός, που μας δίνει πιο γρήγορο πρόγραμμα, ονομάζεται πολυμορφισμός **χρόνου μεταγλώττισης** (compile time) σε αντιδιαστολή με τον «πλήρη» πολυμορφισμό που είναι **χρόνου εκτέλεσης** (run-time). Αλλά όπως καταλαβαίνεις μπορεί να δημιουργήσει πολύ σοβαρά προβλήματα στο πρόγραμμά σου· μην τον χρησιμοποιείς! Γενικώς, όπως μας συμβουλεύει το (CERT 2009):[14]

♦ ***Μην «κρύβεις» κληρονομούμενες μεθόδους που δεν είναι "virtual".***

23.12.2 * Υπερίσχυση ή Επιφόρτωση

Μέχρι τώρα ξέραμε την *επιφόρτωση* (overloading) συναρτήσεων· στο κεφάλαιο αυτό μάθαμε την *υπερίσχυση* (overriding) που είναι κάτι τελείως διαφορετικό. Αν τις μπερδέψεις μπορεί να έχεις προβλήματα, όπως φαίνεται στο παρακάτω παράδειγμα.

Επιστρέφουμε ξανά στο πρόγραμμα της §23.3 και κάνουμε μερικές αλλαγές, στη *B*:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B
```

και στη *D1*:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
private:
   int md1;
}; // D1
```

Η *f()* της *D1 υπερισχύει* –στα αντικείμενα κλάσης *D1*– της *f()* που ορίσαμε στη *B*. Αν βάλουμε στη `main`

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   B b( d );
   cout << b.f() << endl;
```

θα πάρουμε:

```
D1::f()
3
B::f()
2
```

Ας πούμε τώρα ότι αλλάζουμε τη *D1* ως εξής:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
```

[14] Η σύσταση *OBJ02* λέει: «*Do not hide inherited non-virtual member functions*».

```
private:
   int md1;
}; // D1
```

και στη **main** ζητούμε:

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Πρόσεξε τι έχουμε εδώ: Στη βασική κλάση υπάρχει μια μέθοδος

```
   virtual int f() const;
```

και στην παράγωγη μια:

```
   virtual void f( int& aV, int& aDV ) const;
```

Τι σχέση έχουν οι δύο μέθοδοι; Καμιά, απλώς έχουν το ίδιο όνομα. Δηλαδή, στην παράγωγη κλάση, που κληρονομεί την *f()* της *B*, έχουμε *επιφόρτωση*, Τα "**virtual**" δεν παίζουν οποιοδήποτε ρόλο, τουλάχιστον σε ό,τι βλέπουμε!

Δυστυχώς, ο μεταγλωττιστής θα βγάλει πρόβλημα στην:

```
   cout << d.f() << endl;
```

(«**Error E2193 t0.cpp 63: Too few parameters in call to 'D1::f(int &,int &) const' in function main()**»[15]) που, με απλά λόγια, οφείλεται στο εξής: ο μεταγλωττιστής θα αναζητήσει μέθοδο *f()* στη *D1*. Μόλις βρει αυτήν που δηλώνεται στην κλάση θα σταματήσει και δεν θα συνεχίσει την αναζήτηση στη βασική κλάση.

Το πρόβλημα λύνεται αν γράψουμε:

```
   cout << d.B::f() << endl;
```

οπότε και παίρνουμε:

```
B::f()
2
void D1::f()
1  2
```

Ας αλλάξουμε τώρα την *D1* ξαναβάζοντας και την *f()* που είχαμε αρχικώς.

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

Τώρα:

- Η "**virtual int f() const**" *υπερισχύει* της *f()* της βασικής αφού έχει ακριβώς την ίδια δήλωση με εκείνη. Τώρα, το "**virtual**" (στην *f()* της *B*) είναι απαραίτητο!
- Η **virtual void f(int& aV, int& aDV) const** είναι *επιφόρτωση* της προηγούμενης. Το "**virtual**" είναι χρήσιμο μόνο στην περίπτωση που θα γράψουμε συνάρτηση που θα κληρονομήσει τη *D1* (για παράδειγμα τη *DD1*).

Οι

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
```

[15] Borland BC++, v.5.5.

```
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

δίνουν, χωρίς πρόβλημα:

```
D1::f()
3
void D1::f()
1  2
```

Η συμβουλή λοιπόν είναι: *Δώσε άλλο όνομα στη νέα συνάρτηση·* δεν τελείωσαν τα ονόματα!... Γενικώς:

- ***Απόφευγε να επιφορτώνεις κληρονομούμενες ή υπερισχύουσες (virtual) μεθόδους.***

Σημείωση: ▶

Τώρα, που τα κάναμε όλα σωστά, δοκιμάζουμε το εξής:

```
   B* pD( new D1(1,2) );
   cout << pD->f() << endl;
   int a1, a2;
   pD->f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Ο μεταγλωττιστής μας λέει ότι «η *B* δεν έχει συνάρτηση που να ταιριάζει με την `pD->f(a1, a2)`»!

Αυτό είναι ένα «χαλί» για να περάσουμε στην επόμενη παράγραφο.◀

23.13 Δυναμική Τυποθεώρηση – RTTI

Δες άλλο ένα πρόβλημα παρόμοιο με αυτό που είδαμε στη *Σημείωση* της προηγούμενης παραγράφου. Στην §23.9, όπου είχαμε:

```
   Date* pind[2];
// . . .
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

μετά την

```
   pind[1]->forward( 10 );
```

ζητήσαμε

```
   cout << pind[1]->getDay() << endl;
```

και –μετά τη διόρθωση με το "`virtual`"– είδαμε ότι η "`pind[1]->getDay()`" μας έδωσε "`3`". Δεν δοκιμάσαμε όμως να δούμε αν αυξήθηκαν τα δευτερόλεπτα σε "`43`". Τώρα που τα ρυθμίσαμε όλα ας σιγουρευτούμε ότι η *forward()* προχώρησε σωστά τα δευτερόλεπτα. Ζητούμε:

```
   cout << pind[1]->getSec() << endl;
```

και ο μεταγλωττιστής μας έχει μια νέα έκπληξη: «`'class Date' has no member named 'getSec'`». Σοκ για τους απρόσεκτους! "`protected`" βάλαμε, "`virtual`" βάλαμε, τι δεν βάλαμε; Όποιος(-α) μελετάει προσεκτικά ξέρει πως αυτά είναι άσχετα με το πρόβλημα:

- Το "`protected`" δίνει πρόσβαση στις μεθόδους της παράγωγης προς μέλη της βασικής.
- Το "`virtual`" δίνει δυνατότητα επιλογής της σωστής *πολυμορφικής* μεθόδου.

Εδώ τι συμβαίνει; Η *getSec()* δεν είναι πολυμορφική (**virtual**) ώστε να δημιουργήσει πρόβλημα επιλογής μεθόδου· είναι απλώς μέθοδος της παράγωγης κλάσης. Θα πρέπει εσύ να βάλεις στο πρόγραμμά σου τις εντολές που θα ερευνήσουν τον τύπο του αντικειμένου που δείχνει το βέλος και να οδηγήσουν στη σωστή εκτέλεση. Πώς διορθώνουμε τα πράγματα; Με **δυναμική τυποθεώρηση** (dynamic casting):

```
   cout << dynamic_cast<DateTime*>(pind[1])->getSec() << endl;
```

Η `dynamic_cast<`*τύπος βέλους*`>(`τιμή-βέλος`)` εξετάζει αν το όρισμα (τιμή-βέλος) δείχνει στην πραγματικότητα μια τιμή του τύπου βέλους που αναφέρεται ως παράμετρος. Αν είναι επιστρέφει βέλος αυτού του τύπου αλλιώς επιστρέφει "`0`" (`NULL`).

Στην περίπτωσή μας αφού βεβαιωθεί ότι το pind[1] δείχνει αντικείμενο κλάσης *DateTime* μας επιστρέφει βέλος τύπου *DateTime** προς το pind[1].

Εδώ εμείς ξέρουμε ότι η μετατροπή του βέλους pind[1] σε βέλος προς αντικείμενο *DateTime* γίνεται σίγουρα (δεν θα πάρουμε "0") και ύστερα από αυτό μπορούμε να ζητήσουμε τη *getSec()*. Λέμε ότι ο τύπος βέλους προς αντικείμενο της βασικής κλάσης είναι **μετατρέψιμος** (convertible) σε τύπο βέλους προς αντικείμενο παράγωγης κλάσης. Αν δεν είχαμε αυτήν τη σιγουριά θα έπρεπε να δουλέψουμε ως εξής:

```
DateTime* temp( dynamic_cast<DateTime*>(pind[1]) );
if ( temp != 0 )
   cout << temp->getSec() << endl;
```

Όπως βλέπεις, η δυναμική τυποθεώρηση βοηθάει να βρούμε απάντηση στο ερώτημα:

- *Κάποιο συγκεκριμένο βέλος δείχνει αντικείμενο κάποιας συγκεκριμένης κλάσης (ή γενικότερα μεταβλητή κάποιου τύπου);*

Αυτό το ερώτημα δεν μπορούμε να το αποφύγουμε όταν χρησιμοποιούμε αυτά που μάθαμε στις προηγούμενες παραγράφους: βέλη της βασικής κλάσης προς αντικείμενα παράγωγης (ή παραγώγων).

Στο παρακάτω κομμάτι προγράμματος βλέπουμε διάφορες περιπτώσεις χρήσης αυτής της τεχνικής:

```
Date* pd = new DateTime( 2004, 4, 4, 4, 44, 44 );
if ( dynamic_cast<Date*>(pd) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd) == 0 )  cout << "OXI"<<endl;
                                    else  cout << "NAI"<<endl;
Date* pd0 = new Date( 2005, 5, 5 );
if ( dynamic_cast<Date*>(pd0) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd0) == 0 )  cout << "OXI"<<endl;
                                    else  cout << "NAI"<<endl;
```

Αποτέλεσμα:

```
NAI
NAI
NAI
OXI
```

Το *pd*, παρ' όλο που έχει δηλωθεί ως βέλος προς αντικείμενο κλάσης *Date*, δείχνει προς αντικείμενο κλάσης *DateTime*. Έτσι,

- Στη δεύτερη περίπτωση, η `dynamic_cast<DateTime*>(pd)` επιστρέφει ως τιμή βέλος τύπου `DateTime*`.
- Στην πρώτη περίπτωση επιστρέφει το ίδιο το *pd*, που δείχνει προς το υποαντικείμενο κλάσης *Date* του αντικειμένου κλάσης `DateTime` που δείχνει το *pd*.
- Πρόσεξε την τελευταία περίπτωση: το *pd*0 δείχνει αντικείμενο κλάσης *Date*. Επομένως δεν μπορεί να μετατραπεί σε βέλος προς αντικείμενο της παράγωγης κλάσης και η `dynamic_cast<DateTime*>(pd0)` επιστρέφει 0.

Παρατήρηση: ▶

Αν έχουμε δηλώσει:

```
DateTime* pdt( new DateTime(2004, 4, 4, 4, 44, 44) );
```

η "`dynamic_cast<Date*>(pdt) != 0`" είναι η διατύπωση σε C++ της ερώτησης: «*δείχνει το pdt αντικείμενο κλάσης Date;*» Φυσικά η απάντηση είναι "`NAI`" και αυτό βγάζει η

```
if ( dynamic_cast<Date*>(pdt) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
```

Αν ξαναγυρίσεις στην §23.2 μπορείς να πεις ότι σε ένα αντικείμενο κλάσης *DateTime* το *υποαντικείμενο κλάσης Date** δίνεται από την παράσταση

"`*(dynamic_cast<Date*>(this))`"◀

Εκτός από τη δυναμική τυποθεώρηση, **εξακρίβωση τύπου κατά την εκτέλεση** (Run Time Type Identification, RTTI) μπορεί να γίνει και με τον τελεστή `typeid`. Ας ξαναθυμηθούμε μερικά πράγματα που μάθαμε στο *Μέρος Α*, §2.8.1: «*μπορείς να δώσεις:*

`typeid(` *παράσταση* `).name()`

και να πάρεις τον τύπο του αποτελέσματος της παράστασης. Στο πρόγραμμά σου θα πρέπει να περιλάβεις (`#include`) *το "*`typeinfo`*".*» Μην αμφιβάλλεις λοιπόν ότι αν δώσεις:

```
cout << typeid(*pd).name() << endl;
cout << typeid(*pd0).name() << endl;
```

θα πάρεις:

```
DateTime
Date
```

Ο τελεστής "`typeid`" επιστρέφει ένα αντικείμενο κλάσης *type_info*, για την οποίαν έχουν επιφορτωθεί και οι τελεστές "`==`" και "`!=`". Έτσι, αν δώσουμε:

```
if ( typeid(*pd) == typeid(Date) )  cout << "NAI" << endl;
                                 else  cout << "OXI" << endl;
if ( typeid(*pd) == typeid(DateTime) )  cout << "NAI" << endl;
                                    else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(Date) )  cout << "NAI" << endl;
                                  else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(DateTime) )  cout << "NAI" << endl;
                                     else  cout << "OXI" << endl;
```

θα πάρουμε:

```
OXI
NAI
NAI
OXI
```

23.13.1 Μετατροπή Αναφορών

Ας πούμε ότι έχουμε:

```
DateTime dt( 2014, 3, 3, 3, 33, 33 );
cout << "dt = " << dt << endl;
Date& rd( dt );
cout << "rd = " << rd << endl;
```

Αφού όπου μπορούμε να βάλουμε αντικείμενο κλάσης *Date* μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης *DateTime* γράψαμε την εντολή "`Date& rd(dt)`". Αυτές οι εντολές θα δώσουν:

```
dt = 3.3.2014 3:33:33
rd = 3.3.2014
```

«Ε, φυσικά έχουμε τεμαχισμό!» θα πεις. Όχι! Δεν έχουμε τεμαχισμό, αφού δεν έχουμε κάποια αντιγραφή τιμής. Να θυμίσουμε ότι στην §13.4 λέγαμε ότι τα δύο ονόματα –στην περίπτωσή μας *dt* και *rd*– *καθορίζουν την ίδια θέση της μνήμης*. Αυτό που βλέπουμε ως τιμή της *rd* οφείλεται στο ότι γράφεται με τον "`<<`" της *Date*. Πράγματι, αν βάλουμε:

```
DateTime* pRd( reinterpret_cast<DateTime*>(&rd) );
cout << "*pRd = " << *pRd << endl;
```

θα πάρουμε:

```
*pRd = 3.3.2014 3:33:33
```

Θα πεις τώρα «Θα μου τύχει ποτέ να γράψω τέτοια πράγματα στο πρόγραμμά μου;» Όχι βέβαια, αλλά μπορεί να σου τύχει να γράψεις μια συνάρτηση σαν την:

```
void aFunc( Date& aDate )
```

και να θέλεις

- αν κληθεί με όρισμα κλάσης *DateTime* να εκτελεσθούν οι εντολές *EDT* ενώ
- αν κληθει με όρισμα κλάσης *Date* να εκτελεσθούν οι εντολές *ED*.

Ο πάγιος τρόπος χειρισμού του προβλήματος δεν βασίζεται στην ερμηνευτική τυποθεώρηση αλλά στη δυναμική που μπορούμε να την κάνουμε όχι μόνο σε βέλη αλλά και σε τύπους αναφοράς:

```
void aFunc( Date& aDate )
{
// . . .
   try
   {
      DateTime& aDT( dynamic_cast<DateTime&>(aDate) );
   // εντολές EDT για την επεξεργασία του aDT
   }
   catch( bad_cast& )
   {
   // εντολές ED για την επεξεργασία του aDate
   }
// . . .
}
```

Εδώ προσπαθούμε να δούμε –μέσω του τοπικού αντικειμένου *aDT*– την παράμετρο αναφοράς *aDate* ως αντικείμενο κλάσης *DateTime*.

Αν αποτύχει η προσπάθεια ο τελεστής `dynamic_cast` θα ρίξει εξαίρεση κλάσης (*std*::) *bad_cast*. Στο πρόγραμμά σου θα πρέπει έχεις βάλει "`#include <typeinfo>`" όπου υπάρχει ο ορισμός της *bad_cast*.

Εδώ έχουμε μια περίπτωση που η δομή **try/catch** για έγερση και διαχείριση εξαιρέσεων χρησιμοποιείται σαν δομή **if-else**!

23.14 "is_a" ή "has_a"

Στη συνέχεια θα εξετάσουμε ακροθιγώς μερικά σημεία που έχουν σχέση, κατά κύριο λόγο, με αντικειμενοστρεφή σχεδίαση.

Ένας απλουστευτικός τρόπος να δούμε την κληρονομιά είναι και ο εξής: Περιλαμβάνουμε σε κάθε αντικείμενο της παράγωγης ένα αντικείμενο της βασικής. Αυτό όμως μπορούμε να το πετύχουμε πιο απλά. Για παράδειγμα, αντί για:

```
class OfferedCourse :  public Course
{
public:
// . . .
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

θα μπορούσαμε να έχουμε:

```
class OfferedCourse
{
public:
//   . . .
private:
   Course       ocC;
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

Στην περίπτωση αυτή λέμε ότι τα αντικείμενα των δύο κλάσεων συνδέονται με σχέση `has_a` (*OfferedCourse* `has_a` *Course*).

Ποιος τρόπος είναι προτιμότερος; Στο Project 6 ξαναγράφουμε με τον πρώτο τρόπο αυτά που είδαμε στο Project 4. Προσπάθησε να ξαναγράψεις έστω και μέρος του προγράμματος με τον δεύτερο τρόπο και θα καταλάβεις ότι ο πρώτος τρόπος είναι σαφώς προτιμότερος.

Εδώ θα σου δώσουμε ένα πιο απλό παράδειγμα. Έστω ότι έχουμε:

```
class B
```

```
{
public:
   B( int newMb=0 )
      :  mb( newMb ) { };
   virtual ~B() { };
   void f() {  cout << "this is base f" << endl;  };
   virtual void display( ostream& tout )
   {  tout << "base display: mb = " << mb;  };
   virtual void swap( B& other )
   {  std::swap( mb, other.mb );  }
private:
   int mb;
}; // B

class D1 :  public B
{
public:
   D1( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   virtual void swap( D1& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D1
```

Αν δώσουμε:

```
   D1  x( 3, 5 ), u( 19, 17 );
   x.f();
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
   x.B::display( cout );  cout << endl;
   x.swap( u );
   cout << "after swap" << endl;
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
```

θα πάρουμε:

```
this is base f
x: base display: mb = 3   derived display: md = 5
u: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
x: base display: mb = 19   derived display: md = 17
u: base display: mb = 3   derived display: md = 5
```

Αν θέλουμε να έχουμε την ίδια λειτουργικότητα χωρίς κληρονομιές θα πρέπει να έχουμε:

```
class D2
{
public:
   D2( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   ~D2() { };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  bo.f();  };
   void bdisplay( ostream& tout ) {  bo.display( tout );  };
   void swap( D2& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
private:
```

```
   B   bo;
   int md;
}; // D2
```

Όπως βλέπεις, πρέπει να εξοπλίσουμε τη *D2* με δύο ενδιάμεσες συναρτήσεις –*f()* και *bdisplay()*– που επιτρέπουν την πρόσβαση σε μεθόδους της *B*. Θα πάρουμε τα ίδια αποτελέσματα με την πρώτη περίπτωση δίνοντας:

```
   D2 y( 3, 5 ), t( 19, 17 );
   y.f();
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
   y.bdisplay( cout );  cout << endl;
   y.swap( t );
   cout << "after swap" << endl;
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
```

Παρατήρηση:▶

Αν, παραβιάζοντας τον κανόνα που έχουμε βάλει, μεταφέρουμε τη δήλωση "`B bo;`" στην περιοχή "`public`" τα πράγματα απλουστεύονται κάπως:

```
class D3
{
public:
   D3( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void swap( D3& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
   B   bo;
private:
   int md;
}; // D3
```

Δίνοντας:

```
   D3 z( 3, 5 ), v( 19, 17 );
   z.bo.f();
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   z.bo.display( cout );  cout << endl;
   z.swap( v );
   cout << "after swap" << endl;
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
```

παίρνουμε τα ίδια αποτελέσματα που πήραμε και πιο πάνω.◀

Ένα καίριο πρόβλημα που δεν φαίνεται στο παράδειγμά μας είναι το εξής: ενώ μπορούμε να δηλώσουμε "`B* p(new D1)`" δεν μπορούμε να δηλώσουμε "`B* p(new D2)`". Στην πρώτη περίπτωση το βέλος προς αντικείμενο της παράγωγης κλάσης "`new D1`" μετατρέπεται σε βέλος προς αντικείμενο της βασικής και στη συνέχεια μπορούμε να βρούμε τον τύπο του αντικειμένου (που δείχνει το βέλος) με δυναμική τυποθεώρηση ή τον τελεστή "`typeid`". Τι θα κάνεις αν προσπαθήσεις να μετατρέψεις την κλάση *CourseCollection* (αλλά και τη *StudentCollection*) του Project 6; Εκεί υπάρχει ένας δυναμικός πίνακας –ο ccArr– που κάθε στοιχείο του είναι τύπου "`Course*`" αλλά μπορεί να δείχνει και αντικείμενο κλάσης *OfferedCourse*. Πώς θα τον αλλάξεις; Μήπως θα μπορούσαμε να χρησιμοποιήσουμε βέλη "`void*`" και ερμηνευτική τυποθεώρηση; Ναι, βέβαια, αλλά δεν είναι και τόσο απλό. Σε ένα τέτοιο βέλος δεν μπορείς να κάνεις απόπαραπομπή αν δεν κάνεις προηγουμένως τυποθεώρηση. Σε ποιον τύπο όμως; Θα πρέπει κάπου να έχεις φυλαγμένη τη σχετική πληροφορία…

Η σχέση "`is_a`" μπορεί να μας λύσει προβλήματα με απλό τρόπο αλλά, από την άλλη μεριά, *οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και*

τη σωστή νοηματική σχέση. Ας πούμε ότι έχεις να γράψεις μια κλάση *Computer* που περιγράφει έναν υπολογιστή ενώ έχεις ήδη μια κλάση *HardDisk* που περιγράφει έναν σκληρό δίσκο. *Μην ξεκινήσεις* με τη λογική *Computer* `is_a` *HardDisk* –ώστε ένα αντικείμενο *Computer* να περιλαμβάνει ένα αντικείμενο *HardDisk*– επειδή συνειδητοποίησες ότι οι διεπαφές (περιοχές "`public`") μοιάζουν πολύ. Η σωστή σχέση είναι *Computer* `has_a` *HardDisk*.

Στη σχέση "`has_a`" θα επανέλθουμε.

Πάντως αν περιορίσεις τις επιλογές σου μεταξύ "`is_a`" και "`has_a`" μπορεί να οδηγηθείς σε σχεδιαστικά λάθη:

Παράδειγμα

Ας πούμε ότι προσθέτουμε στην *OfferedCourse* ένα μέλος επιπλέον, ας το πούμε *ocSemester*, για να κρατούμε και το εξάμηνο που προσφέρθηκε το μάθημα. Αν τώρα, σε ένα πρόγραμμα κάνουμε συγκριτική επεξεργασία της διεξαγωγής των μαθημάτων για τα τελευταία έξη εξάμηνα θα έχουμε εξάδες αντικειμενων *OfferedCourse* που θα έχουν το ίδιο υποαντικείμενο *Course*. Για να το απόφύγουμε μπορούμε

- να βάλουμε στην *OfferedCourse* απλώς ένα βέλος προς αντικείμενο κλάσης *Course* ή
- να βάλουμε στην *OfferedCourse* μόνον τον κωδικό μαθήματος.

Αυτή είναι μια περίπτωση **αντιστοίχισης** (association) **πολλά-προς-ένα** (many-to-one) ή $N:1$· ούτε "`is_a`" ούτε "`has_a`".

Καλή λοιπόν η κληρονομιά, αλλά δεν είναι κατ' ανάγκη η καλύτερη επιλογή πάντοτε· χρειάζεται λίγη σκέψη πριν τη χρησιμοποιήσουμε.

23.15 "private", "protected" και "public"

Γενικότερα, όταν σχεδιάζεις μια κλάση θα πρέπει να σκεφτείς καλά πριν αποφασίσεις τι θα βάλεις **private**, τι θα βάλεις **protected** και τι θα βάλεις **public**.

- ***Αν ένα μέλος κλάσης είναι*** `private` ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης και β) φίλες κλάσεις ή συναρτήσεις της κλάσης.***
- ***Αν ένα μέλος είναι*** **protected** ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης β) φίλες συναρτήσεις της κλάσης γ) μεθόδους των παραγώγων κλάσεων της και δ) φίλες συναρτήσεις παραγώγων κλάσεων.***
- ***Αν ένα μέλος είναι*** **public** ***μπορεί να χρησιμοποιηθεί από οποιαδήποτε συνάρτηση.***

Όταν γράφεις μια παράγωγη κλάση θα πρέπει να αποφασίσεις για τον τρόπο που αυτή κληρονομεί τη βασική:

- ***Αν δηλώσεις "*`class D: private B`*" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***).***
- ***Αν δηλώσεις "*`class D: protected B`*" τότε τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από α) τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) τις μεθόδους και τις φίλες των παραγώγων κλάσεων της D. Δικαίωμα να μετατροπής ενός*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***) έχουν μόνον α) οι μεθοδοι της D και οι φίλες συναρτήσεις της D και β) οι μέθοδοι και τις φίλες των παραγώγων κλάσεων της D.***
- ***Αν δηλώσεις "*`class D: public B`*" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από οποιαδήποτε συνάρτηση β) οποιαδήποτε συνάρτηση μπορεί να μετατρέψει ένα βέλος*** `D*` ***σε*** `B*` ***και μια αναφορά*** `D&` ***σε*** `B&`***.***

Ας συμπληρώσουμε λοιπόν τον κανόνα που δώσαμε πιο πάνω:

- ***Αν έχουμε δηλώσει "`class D: public B`" τότε όπου μπορούμε να βάλουμε, στο πρόγραμμά μας, ένα αντικείμενο κλάσης B μπορούμε να βάλουμε και αντικείμενο κλάσης D.***

Αυτή είναι η ακριβέστερη διατύπωση της *δυνατότητας υποκατάστασης* που είδαμε στην §23.2.

Η κληρονομιά "`private`" είναι στην πραγματικότητα ένας τρόπος υλοποίησης της συσχέτισης "`has_a`" που είδαμε στην προηγούμενη παράγραφο. Στο πρόγραμμα που γράψαμε εκεί βάζουμε μια ακόμη κλάση:

```
class D4 : private B
{
public:
   D4( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D4() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  B::f();  };
   void bdisplay( ostream& tout ) {  B::display( tout );  };
   void swap( D4& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D4
```

Οι εντολές:

```
   D4 v( 3, 5 ), w( 19, 17 );
   v.f();
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
   v.bdisplay( cout );  cout << endl;
   v.swap( w );
   cout << "after swap" << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
```

δίνουν:

```
this is base f
v: base display: mb = 3   derived display: md = 5
w: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
v: base display: mb = 19   derived display: md = 17
w: base display: mb = 3   derived display: md = 5
```

Να παρατηρήσουμε τα εξής:

- Λέμε παραπάνω: «*τα μέλη της B που είναι "`public`" ή "`protected`" μπορεί να χρησιμοποιηθούν από τις μεθόδους της D.*» Με βάση αυτήν τη δυνατότητα γράφουμε τις μεθόδους *D4::f()*, *D4::bdisplay()* και *D4::swap()*. Δεν γίνεται να χρησιμοποιηθούν κάπως «πιο έξω», ας πούμε στη `main()`; Όχι! Αν γράψεις "`v.f();`" ο μεταγλωττιστής θα σου πεί: «`'B::f()' is not accessible in function main()`». Παρόμοια αντιμετώπιση θα βρεις αν απόπειραθείς να γράψεις "`v.B::display(cout);`". Έτσι, χρειάζονται οι ενδιάμεσες συναρτήσεις *D4::f()*και *D4::bdisplay()*, όπως ακριβώς και στη *D2*.
- Λέμε παραπάνω: «*μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα* `D*` *σε* `B*` *(και* `D&` *σε* `B&`*).*» Με βάση αυτήν τη δυνατότητα γράφουμε τη *D4::swap()* που είναι ολόιδια με τη *D1::swap()*. Η κλήση "`B::swap(other);`" απαιτεί μετατροπή μιας αναφοράς *D4&* (*other*) σε *B&* που περιμένει η *B::swap()*. Ούτε αυτή η δυνατότητα μπορεί να χρησιμοποιηθεί «πιο έξω». Αν, ας πούμε, έχεις μια συνάρτηση

```
int f1( B& ap )
```

```
{
//...
```

τότε η κλήση:

```
cout << f1( v );
```

δεν είναι δεκτή· «`Cannot initialize 'B &' with 'D4' in function main()`» θα μας πει ο μεταγλωττιστής.

Ακόμη, δεν μπορείς μέσα στο πρόγραμμά σου να γράψεις:

```
B* bp( new D4 );
```

Όπως καταλαβαίνεις η περίπτωση "`class D: private B`" είναι χρήσιμη όταν θέλεις να χρησιμοποιήσεις τη *B* για την *υλοποίηση* των μεθόδων της *D*. Για τον λόγο αυτόν θα δεις να γράφεται "*D* `is_implemented_in_terms_of` *B*" εκτός από "*D* `has_a` *B*".

Τα ίδια ισχύουν σε μεγάλο ποσοστό και για την περίπτωση "`class D: protected B`". Στην περίπτωση αυτήν παράγωγες κλάσεις της *D* βλέπουν αυτά που κληρονόμησε η *D* από τη *B* ως "`protected`".

23.16 Αφηρημένες Κλάσεις

Αν παρατηρήσουμε τα παραδείγματα που δώσαμε μέχρι τώρα βλέπουμε το εξής: *Σε κάθε παραγωγή η βασική κλάση είναι πιο απλή από την παράγωγη*. Αυτός είναι ένας τρόπος να διαχειριζόμαστε την πολυπλοκότητα στα προγράμματα: ασχολούμαστε με τα προβλήματα σταδιακά, οι δυσκολίες έρχονται μια-μια και όχι όλες μαζί. Ας δούμε άλλο ένα παράδειγμα:

Μια εταιρεία παραγωγής/διανομής ηλεκτρικής ενέργειας έχει τρεις κατηγορίες πελατών:

1. *μεγάλες βιομηχανικές μονάδες που χρεώνονται με* 0.03 *ευρώ/ kWh,*
2. *μικρομεσαίες βιοτεχνικές μονάδες που χρεώνονται με* 0.06 *ευρώ/kWh συν μια πάγια χρέωση που είναι διαφορετική για κάθε καταναλωτή και*
3. *οικιακούς καταναλωτές που χρεώνονται με πάγια χρέωση –διαφορετική για κάθε καταναλωτή– συν* 0.06 *ευρώ/kWh για τις πρώτες A kWh και* 0.1 *ευρώ/kWh για την κατανάλωση πέραν της A. Η A καθορίζεται χωριστά για κάθε καταναλωτή.*

Κάθε καταναλωτής, άσχετα από την κατηγορία του, έχει έναν κωδικό αριθμό (θετικός ακέραιος).

Να ορισθεί ιεραρχία κλάσεων για να παρασταθούν τα στοιχεία των πελατών. Κάθε κλάση θα έχει οπωσδήποτε μέθοδο για τον υπολογισμό της χρέωσης του πελάτη.

Ας ξεκινήσουμε από την πρώτη κατηγορία, για την οποίαν γράφουμε μια κλάση, ας την πούμε *Industrial*. Τι χρειάζεται να κρατούμε για έναν πελάτη αυτής της κατηγορίας; Μόνον τον κωδικό αριθμό. Και τι μεθόδους θα χρειαστούμε; Μια μέθοδο για να παίρνουμε τον κωδικό, μια για να τον ορίζουμε και μια για τη χρέωση.

Πάμε τώρα στη δεύτερη κατηγορία (κλάση *SmallMediumEnt*). Εδώ χρειαζόμαστε όλα όσα έχει η πρώτη κλάση και ακόμη ένα μέλος για την πάγια χρέωση. Φυσικά θα χρειαστούμε και δύο μεθόδους για τον χειρισμός αυτού του μέλους.

Στην τρίτη κατηγορία (κλάση *HomeCons*) θα χρειαστούμε όσα έχει η δεύτερη και ακόμη ένα μέλος για την *A* και δύο μεθόδους για τον χειρισμό της.

Επομένως, βασική κλάση θα είναι η (απλούστερη από όλες) *Industrial*. Η επόμενη απλούστερη είναι η κλάση *SmallMediumEnt* που θα παράγεται από την πρώτη. Η κλάση *HomeCons* είναι η πιο πολύπλοκη και θα είναι παράγωγη *SmallMediumEnt*.

Μέθοδος για τη χρέωση θα υπάρχει και στις τρεις κλάσεις (πολυμορφική). Φυσικά δεν είναι ίδια και για τις τρεις κλάσεις.

Φτάσαμε λοιπόν σε μια ιεραρχία:

Industrial ← *SmallMediumEnt* ← *HomeCons*

Αλλά, μπορούμε να πούμε ότι μια "*μικρομεσαία βιοτεχνική μονάδα*" είναι μια "*μεγάλη βιομηχανική μονάδα*" ή ότι ένας "*οικιακός καταναλωτής*" είναι μια "*μικρομεσαία βιοτεχνική μονάδα*"; Όχι φυσικά! Εκείνο που μπορούμε να πούμε είναι ότι: Από προγραμματιστική άποψη

- ο λογαριασμός μιας "*μικρομεσαίας βιοτεχνικής μονάδας*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μεγάλης βιομηχανικής μονάδας*" και
- ο λογαριασμός ενός "*οικιακού καταναλωτή*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μικρομεσαίας βιοτεχνικής μονάδας*".

Δηλαδή, για να βρούμε τη λογική της ιεραρχίας των κλάσεων που βγάλαμε πρέπει *να προχωρήσουμε κατά ένα επίπεδο αφαίρεσης*.

Στο Σχ. 23-6 βλέπεις μια άλλη ιεραρχία κλάσεων για το παράδειγμά μας. Εισάγουμε μια νέα κλάση, την

```
class ElecConsumer
{
public:
   ElecConsumer( int consCodeP );
   unsigned long int getConsCode() const {  return ecConsCode;  }
   void setConsCode( int consCodeP );
   virtual double charge( double consumption ) const = 0;
protected:
   unsigned long int ecConsCode;
}; // ElecConsumer
```

Πρόσεξε όμως ένα ιδιαίτερο χαρακτηριστικό: Η εικονική μέθοδος *charge* δηλώνεται αλλά δεν ορίζεται· αυτό το δείχνουμε με εκείνο το "`= 0`". Λέμε ότι η *charge* είναι μια **γνήσια εικονική μέθοδος** (true virtual method).

Μια κλάση που έχει μια τουλάχιστον γνήσια εικονική μέθοδο λέγεται **αφηρημένη** (abstract) κλάση. Φυσικά, μια τέτοια κλάση, αφού είναι ελλιπής, δεν μπορεί να έχει αντικείμενα. Σε αντιδιαστολή, οι άλλες κλάσεις, που είναι πλήρεις, και έχουν δικά τους αντικείμενα ονομάζονται **συγκεκριμένες** (concrete) κλάσεις.

Χρησιμοποιούμε τις αφηρημένες κλάσεις για να δημιουργούμε "σωστές" ιεραρχίες κλάσεων, όπου φυσικά μια αφηρημένη κλάση (όπως η *ElecConsumer*) είναι βασική κλάση ενώ οι παράγωγές της είναι συγκεκριμένες. Για τον λόγο αυτόν θα τη δεις και ως **αφηρημενη βασική κλάση**. Σε μια ιεραρχία κλάσεων μπορεί να υπάρχουν αφηρημένες κλάσεις σε περισσότερα από ένα στάδια παραγωγής.

Από νοηματική άποψη η νέα ιεραρχία κλάσεων είναι πολύ καλύτερη από την αρχική. Πράγματι, στην αφηρημένη βασική κλάση *ElecConsumer* υπάρχουν τα γενικά χαρακτηριστικά ενός καταναλωτή ηλεκτρικής ενέργειας. Στη συνέχεια, αντί της

Industrial ← *SmallMediumEnt* ← *HomeCons*

έχουμε τρεις παραγωγές:

ElecConsumer ← *Industrial*

ElecConsumer ← *SmallMediumEnt*

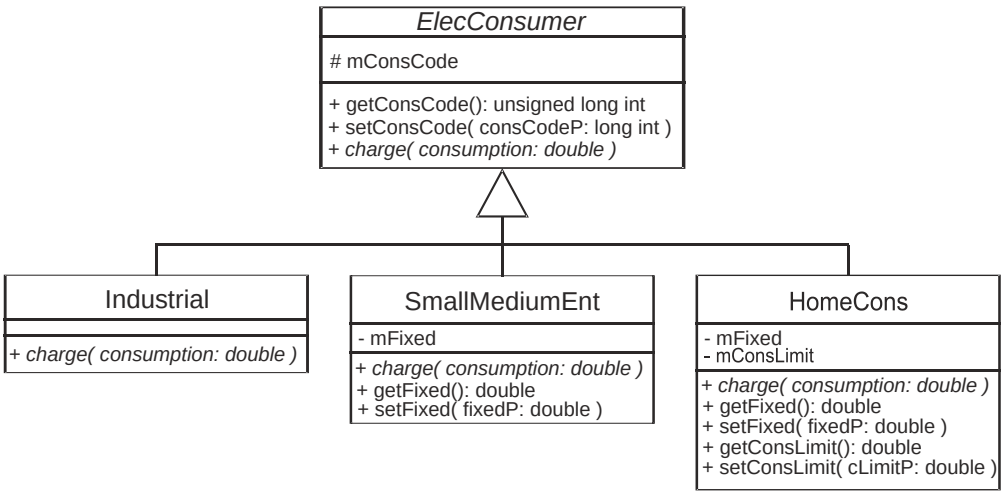
ElecConsumer ← *HomeCons*

που έχουν νόημα: Ο *βιομηχανικός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*, η *μικρομεσαία επιχείρηση* `is_a` *καταναλωτής ηλεκτρικής ενέργειας* και ο *οικιακός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*.

23.17 Η Σειρά Δημιουργίας

Η σειρά δημιουργίας ενός αντικειμένου, που είδαμε εν μέρει στις §3.1.2 και §10.3 μπορεί τώρα να δοθεί ολοκληρωμένη:

- Δημιουργούνται τα υποαντικείμενα των εικονικών βασικών κλάσεων όπως εμφανίζονται στη λίστα κληρονομιάς.



Σχ. 23-6 Οι τρεις αληθινές κλάσεις, *Industrial*, *SmallMediumEnt*, *HomeCons*, κληρονομούν την αφηρημένη βασική κλάση *ElecConsumer*.

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται στην κλάση.
- Εκτελείται το σώμα του δημιουργού.

23.18 Πολλαπλή Κληρονομιά

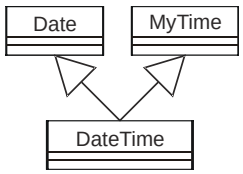
Έστω ότι έχουμε την κλάση:

```
class MyTime
{
friend ostream& operator<<( ostream& tout, const MyTime& rhs );
public:
   MyTime( int hp = 0, int minp = 0, int sp = 0 );
   unsigned int getHour() const {  return mtHour;  };
   unsigned int getMin() const {  return mtMin;  };
   unsigned int getSec() const {  return mtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
   MyTime& operator++();
protected:
   unsigned int mtHour;    // hour (0 .. 23)
   unsigned int mtMin;     // minutes (0 .. 59)
   unsigned int mtSec;     // seconds (0 .. 59)
}; // MyTime
```

Έχουμε επίσης τη γνωστή μας *Date*. Δες έναν άλλον τρόπο να πάρουμε μια κλάση *DateTime*:

```
class DateTime : public Date, public MyTime
{
friend ostream& operator<<( ostream& tout, const DateTime& rhs );
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 )
      : Date( yp, mp, dp ), MyTime( hp, minp, sp ) { };
}; // DateTime

ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return ( tout << rhs.dDay << '.' << rhs.dMonth << '.' << rhs.dYear << ' '
                 << rhs.mtHour << ':' << rhs.mtMin << ':' << rhs.mtSec );
} // operator<< DateTime
```

Δηλαδή η *DateTime* κληρονομεί δύο κλάσεις: τη *Date* και τη *MyTime*. Αυτό συμβολίζεται διαγραμματικώς όπως βλέπεις στο Σχ. 23-7.



Σχ. 23-7 Η *DateTime* κληρονομεί τη *Date* και τη *MyTime*.

Γενικώς μια παράγωγη κλάση μπορεί να κληρονομεί

- πολλές κλάσεις και
- όχι κατ' ανάγκη με τον ίδιο τρόπο, π.χ.:

```
class X : public A, private B, public C { /* ... */ };
```

Αν δηλώσεις:

```
   DateTime dt( 2007, 11, 26, 19, 30, 31 );
```

τότε η *dt* έχει όλες τις ιδιότητες που κληρονόμησε και από τις δύο βασικές κλάσεις! Όλες; Και με τον "++" που ορίζεται και στη *Date* και στη *MyTime* τι γίνεται; Εδώ έχουμε μια περίπτωση **ασάφειας** ή **αμφιβολίας** (ambiguity) η οποία λύνεται έτσι:

```
   dt.Date::operator++();
   cout << dt << endl;
   dt.MyTime::operator++();
   cout << dt << endl;
```

Αποτέλεσμα:

```
27.11.2007 19:30:31
27.11.2007 19:30:32
```

Η πρώτη, `dt.Date::operator++()`, αυξάνει την ημέρα από 26 σε 27 ενώ η δεύτερη, `dt.MyTime::operator++()`, αυξάνει τα δευτερόλεπτα από 31 σε 32.

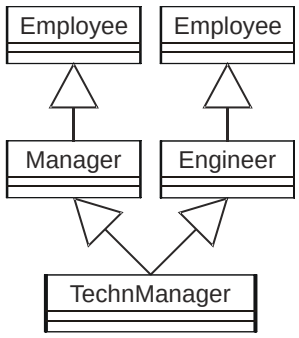
Οι περιπλοκές όμως μπορεί να είναι πιο εντυπωσιακές:

```
class B1 : public L { /* ... */ };
class B2 : public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```

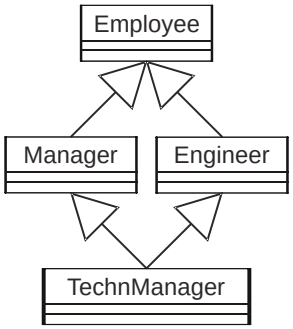
Κάθε αντικείμενο κλάσης *D* έχει δύο υποαντικείμενα κλάσης *L*, το ένα έρχεται από τη *B2* και το άλλο από τη *B3*. Απίθανη περίπτωση; Κάθε άλλο! Βάλε αντί για *L*, *Employee*, αντί για *B1*, *Manager*, αντί για *B2*, *Engineer*, αντί για *D*, *TechnManager* και έχεις το πιο «χειροπιαστό» παράδειγμα στο Σχ.23-8.

Η C++ σου δίνει δυνατότητα να ξεχωρίσεις τα δύο υποαντικείμενα: `B2::L` και `B3::L`. Αλλά το ερώτημα είναι: πόσο χρήσιμο είναι να έχουμε δύο υποαντικείμενα; Στο παράδειγμά μας, του Σχ. 23-8, είναι φανερό ότι για τον Τεχνικό Διευθυντή θέλουμε τα στοιχεία που κρατούμε για κάθε υπάλληλο μια φορά μόνον. Θέλουμε μια εικόνα κληρονομιάς σαν αυτήν του Σχ. 23-9. Αυτό επιτυγχάνεται με την **εικονική** (virtual) κληρονομιά:

```
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```



Σχ. 23-8 Η *TechnManager* κληρονομεί την *Employee* δύο φορές.



Σχ. 23-9 Με κληρονομιά `virtual` η *TechnManager* κληρονομεί την *Employee* μια φορά.

23.19 Διεπαφές

Όπως λέγαμε στην §19.1.3, «*Η οποιαδήποτε επαφή του κάθε προγράμματος με το αντικείμενο γίνεται μέσω αυτών που υπάρχουν στις περιοχές "*`public`*". Αυτά αποτελούν και το τμήμα διεπαφής (interface part) του αντικειμένου.*»

Η Java[16] δίνει την εξής δυνατότητα στον προγραμματιστή: να ορίσει ένα ανεξάρτητο τμήμα διεπαφής και στη συνέχεια να ορίσει τις κλάσεις που το χρησιμοποιούν. Για παράδειγμα, αντιγράφουμε από τα Java Tutorials[17]

```
interface Bicycle
{
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

Αυτή είναι η διεπαφή μιας κλάσης που παριστάνει ποδήλατα. Αν στη συνέχεια θέλουμε μια κλάση για ποδήλατα ACME γράφουμε:

```
class ACMEBicycle implements Bicycle
{
    int cadence = 0;
    int speed = 0;
    int gear = 1;
// . . .
    void speedUp(int increment)
    {   speed = speed + increment;   }
// . . .
}
```

Η C++ δεν μας δίνει αυτήν ακριβώς τη δυνατότητα αλλά μπορούμε να κάνουμε τα ίδια πράγματα χρησιμοποιώντας αφηρημένες κλάσεις. Για το παράδειγμά μας ορίζουμε:

```
class Bicycle
{
public:
   virtual void changeCadence( int newValue ) = 0;
   virtual void changeGear( int newValue ) = 0;
   virtual void speedUp( int increment ) = 0;
   virtual void applyBrakes( int decrement ) = 0;
};

class ACMEBicycle :  public Bicycle
{
public:
   ACMEBicycle()
   {  cadence = 0;  speed = 0;  gear = 1;  }

   virtual void changeCadence( int newValue )
   {  cadence = newValue;  }

   virtual void changeGear( int newValue )
   {  gear = newValue;  }

   virtual void speedUp( int increment )
   {  speed = speed + increment;  }

   virtual void applyBrakes( int decrement )
   {  speed = speed - decrement;  }

   void printStates()
   {  cout << "cadence:" << cadence << " speed:" << speed
```

[16] Παρόμοια δυνατότητα δίνει και η C#.

[17] `http://docs.oracle.com/javase/tutorial/java/concepts/interface.html`

```
            << " gear:" << gear << endl;  }
private:
   int cadence;
   int speed;
   int gear;
}; // ACMEBicycle
```

23.20 Ανακεφαλαίωση

Από μια *βασική κλάση B* μπορούμε να πάρουμε μια *παράγωγη κλάση D* με τη δήλωση:

```
class D :   public B
{ // . . .
```

Η κλάση *D κληρονομεί* τη *B* με την εξής έννοια:

- Όπου μπορούμε να βάλουμε αντικείμενο κλάσης *B* μπορούμε να βάλουμε αντικείμενο κλάσης *D* (*δυνατότητα υποκατάστασης*) αφού:
  - Κάθε αντικείμενο κλάσης *D* έχει όλα τα μέλη και όλες τις συναρτήσεις-μέλη που έχει ένα αντικείμενο κλάσης *B*.
  - Μπορεί να έχει επιπλέον μέλη ή/και συναρτήσεις-μέλη.
- Η κλάση *D κληρονομεί* στατικά μέλη και στατικές συναρτήσεις-μέλη της *B*.
- Δεν κληρονομούνται οι δημιουργοί, ο τελεστής εκχώρησης και ο καταστροφέας της βασικής κλάσης. Μπορείς όμως να τους χρησιμοποιήσεις για να ορίσεις τις αντίστοιχες συναρτήσεις της παράγωγης.

Στην παράγωγη κλάση μπορούμε να ξαναορίσουμε όποιες μεθόδους της βασικής θέλουμε οπότε αυτές *υπερισχύουν* των αντίστοιχων της βασικής. *Στη βασική κλάση οι μέθοδοι που θα ξαναορισθούν πρέπει να είναι δηλωμένες* "`virtual`".

Ορίζοντας τις συναρτήσεις-μέλη της παράγωγης κλάσης μπορεί να χρειαστεί να διαχειριστείς μέλη της βασικής. Ο καλύτερος τρόπος είναι να το κάνεις μέσω των δημιουργών και των μεθόδων "*get*" και "*set*". Αν θεωρείς απαραίτητη την κατ' ευθείαν πρόσβαση στα μέλη δήλωσέ τα (στη βασική) "`protected`" και όχι "`private`".

Λόγω της *δυνατότητας υποκατάστασης*, σε ορισμένες περιπτώσεις χρειάζεται να βρούμε τον τύπο κάποιου αντικειμένου. Η *δυναμική τυποθεώρηση* και ο τελεστής "`typeid`" δίνουν τη λύση σε αυτό το πρόβλημα.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

23-1 Τι θα γράψει το παρακάτω πρόγραμμα:

```
#include <iostream>
class A
{
public:
   A( char c = 'A' )  :  x( c )
   {   cout << "A=" << x; }
   ~A() {  cout << "~A";  }
   A& operator=( const A& a )
   {
      x = a.x;
      cout << "A=" << x;
   }
private:
   char x;
}; // class A
```

```
class C : public A
{
public:
   C( char c = 'C' )  :  z( c )
   {  cout << "C=" << z;  }
   ~C()
   {  cout << "~C";  }
   C& operator=( const C& c )
   {
      A::operator=(c);
      z = c.z;
      cout << "C=" << z;
   }
private:
```

```
class B : public A
{
public:
   B( char c = 'B' )  :  y( c )
   {  cout << "B=" << y;  }
   ~B()
   {  cout << "~B"; }
   B& operator=( const B& b )
   {
      A::operator=( b );
      y = b.y;
      cout << "B=" << y;
   }
private:
   char y;
}; // class B
```

```
   char z;
}; // class C

int main()
{
   cout << "Prints";
   cout << endl << "1:";
   A* b = new B;
   A* c = new C;
   cout << endl << "2:";
   A bb = *b;
   A cc = *c;
   cout << endl << "3:";
   bb = cc;
} // main
```

Β Ομάδα

23-2 Έχουμε τις εξής δύο κλάσεις:

```
class Order
{
public:
. . .
private:
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
}; // Order
```

```
class Auto
{
public:
. . .
private:
   std::string manuf;
   std::string model;
   char        country[3];
   Date        purchDate;
   std::string regNumber;
}; // Auto
```

όπου *Date* η γνωστή κλάση και:

```
struct Good
{
   unsigned int code;
   std::string  unit;
   double       quantity;
   double       price;
}; // Good
```

α) Η πρώτη (`Order`) περιγράφει μια παραγγελία: Ο δυναμικός πίνακας `goods`, με `gCount` στοιχεία, έχει τα αγαθά που παραγγέλνονται. Όπως φαίνεται και από την `struct Good`, για κάθε είδος, έχουμε τον κωδικό του (*code*), τη μονάδα μέτρησης (*unit*) και την ποσότητα (*quantity*) που παραγγέλνεται. Η αξία (*price*) δεν χρησιμοποιείται. Στην `oDate` έχουμε την ημερομηνία της παραγγελίας ενώ έχουμε και έναν (μοναδικό) αριθμό παραγγελίας (`orderNum`).

β) Η δεύτερη (`Auto`) περιγράφει ένα αυτοκίνητο: τον κατασκευαστή (`manuf`), το μοντέλο (`model`), τη χώρα κατασκευής (`country`, κωδικός χώρας με δύο γράμματα), ημερομηνία αγοράς (`purchDate`), αριθμό (`regNumber`).[18]

Για κάθε μια από τις δύο κλάσεις, άσχετα από την προβλεπόμενη χρήση, παράθεσε (χωρίς υλοποίηση) τους δημιουργούς, τον καταστροφέα και τις μεθόδους / τελεστές που πρέπει να υλοποιήσουμε. Για κάθε ένα από αυτά θα δικαιολογείς γιατί χρειάζεται ή γιατί δεν χρειάζεται η υλοποίηση.

[18] Προφανώς και για τις δύο κλάσεις μπορούμε να σκεφτούμε και άλλα μέλη, απαραίτητα για διάφορες εφαρμογές. Εδώ, θα απαντήσεις τις ερωτήσεις που ακολουθούν με βάση τα μέλη που δίνονται.

Άσχετα από την απάντησή σου στην προηγούμενη ερώτηση γράψε τους δημιουργούς ερήμην και αντιγραφής και για τις δύο κλάσεις. Σχολίασε αυτά που έγραψες και με βάση τις απαντήσεις που έδωσες στην προηγούμενη ερώτηση.

Για την κλάση **Auto** υλοποίησε τον τελεστή εγγραφής ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**).

Για την κλάση **Order** υλοποίησε μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

23-3 Η κλάση:

```
class Invoice
{
public:
. . .
private:
   unsigned int invoiceNum;
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
   Date         iDate;
   double       total;
}; // Order
```

παριστάνει ένα τιμολόγιο. Έχει τον αριθμό της παραγγελίας στην οποίαν αναφέρεται αλλά και δικό του αριθμό τιμολογίου (**invoiceNum**). Έχει την ημερομηνία της παραγγελίας αλλά και την ημερομηνία έκδοσης του τιμολογίου (**iDate**). Τα αγαθά περιγράφονται με τον ίδιο τρόπο, όπως και στην παραγγελία, αλλά τώρα είναι συμπληρωμένη και η αξία (*price*). Τέλος, υπάρχει και η συνολική αξία (*total*).

Όπως έκανες για την **Order**, υλοποίησε και για την **Invoice** μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

Η **Invoice** μπορεί να θεωρηθεί ως παράγωγη της **Order**. Γράψε τη δήλωσή της ώστε να φαίνεται αυτή σχέση.

Στην **Order** και στις μεθόδους που έγραψες γι' αυτήν κάνε όλες τις αλλαγές που απαιτούνται ώστε η κληρονόμηση να γίνεται σωστά.

23-4 Έχουμε τις κλάσεις:

```
class A
{
public:
. . .
protected:
   double*      g;
   unsigned int nElmn;
   std::string  c;
}; // A
```

```
class B: public A
{
public:
. . .
private:
   double r;
}; // B
```

(η *nElmn* δείχνει το πλήθος στοιχείων του πίνακα g.)

Συμπλήρωσέ τις με ό,τι χρειάζεται ώστε

α) να μπορώ να γράψω τα εξής:

```
int main()
{
   double q[3] = { 3.0, 1.5, -1.1 };
   A a1;
// a1.nElmn==0 && a1.g==0 && c==""
   A a2( 3, q, "an object" );
// a2.nElmn==3 && a2.g[0]==3.0 && a2.g[1]==1.5 && a2.g[2]==-1.1 &&
// a2.c=="an object"
   A a3( a2 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==1.5 && a3.g[2]==-1.1 &&
```

```
// a3.c=="an object"

   a3.setG( 1, 5.5 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==5.5 && a3.g[2]==-1.1 &&
// a3.c=="an object" &&
// a2.g[1]==1.5 && q[1]==1.5

   B b1( a3, 7.35 );
// b1.nElmn==3 && b1.g[0]==3.0 && b1.g[1]==5.5 && b1.g[2]==-1.1 &&
// b1.c=="an object" && b1.r==7.35
   . . .
```

Στα σχόλια που ακολουθούν κάθε εντολή δίνεται η κατάσταση του θα έχουμε μετά την εκτέλεσή της.

β) Ακόμη, αν έχω δηλώσει:

```
   A*      z[2] = { &a2, &b1 };
   fstream tout( "abc.dta", ios_base::out|ios_base::binary );
```

οι εντολές:

```
   z[0]->write( tout );
   z[1]->write( tout );
```

θα πρέπει να έχουν ως αποτέλεσμα την εγγραφή στον αρχείο `abc.dta`, σε εσωτερική παράσταση (binary), των τιμών των *a*2, *b*1 με την εξής σειρά:

```
3, 3.0, 1.5, -1.1, an object, 3, 3.0, 5.5, -1.1, an object, 7.35
```

23-5 Θέλουμε να γράψουμε μια κλάση:

```
class LongBitMap
{
public:
   typedef unsigned long int  ULong;
   LongBitMap()  {  mBm = new ULong[1];  mBm[0] = 0;
                    mNoOfBits = 0;  mArrSize = 1;  }
   LongBitMap( LongBitMap& other );
   ~LongBitMap() {  delete [] ULong;  }
   long getNoOfBits() const {  return mNoOfBits;  }
   LongBitMap& operator=( LongBitMap& other );
   LongBitMap& operator&=( LongBitMap& other );
   LongBitMap& operator|=( LongBitMap& other );
   LongBitMap& operator^=( LongBitMap& other );
   void lSetBit( int pos );
   void lClearBit( int pos );
   ULong lCount1() const;
   int lBitValue( int pos ) const;
private:
   ULong* mBm;          // δυναμικός ψηφιοπίνακας
   ULong  mNoOfBits;    // πλήθος των bits που χρησιμοποιούμε
   ULong  mArrSize;     // πλήθος στοιχείων (τύπου ULong) του mBm
}; // class LongBitMap
```

για τη διαχείριση μεγάλων ψηφιοπινάκων. Όπως βλέπεις, ο ψηφιοπίνακας παριστάνεται με έναν δυναμικό πίνακα με στοιχεία τύπου `unsigned long int`. Στο mBm[0] υπάρχουν τα bits 0 μέχρι 31, στο mBm[1] τα bits 32 μέχρι 63, στο mBm[2] τα bits 64 μέχρι 95 κ.ο.κ. Το μέλος *mNoOfBits* μας δίνει το πλήθος bits που χρησιμοποιούνται. Το μέλος *mArrSize* μας δίνει το πλήθος των στοιχείων του δυναμικού πίνακα mBm.

Εσύ θα πρέπει να βοηθήσεις στην ολοκλήρωση της υλοποίησης. Να οι προδιαγραφές των μεθόδων που πρέπει να γράψεις:

Δημιουργός Αντιγραφής.

Τελεστής ψηφιακού "**and**" (`operator&=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Τελεστής ψηφιακού "**xor**" (`operator^=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Μέθοδος *lSetBit*. Βάζει 1 στο bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 τότε ο πίνακας επεκτείνεται (όπως μάθαμε στη *renew*) ώστε να περιέχει και θέση *pos*.

Μέθοδος *lBitValue*. Επιστρέφει την τιμή του bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 ρίχνει εξαίρεση.

Μπορείς να χρησιμοποιήσεις χωρίς να αντιγράψεις εδώ τα περιγράμματα που συναρτήσεων για ψηφιοχάρτες που υπάρχουν στις σημειώσεις.

project

6

Φοιτητές και Μαθήματα με Κληρονομιές

Περιεχόμενα:

Prj06.1 Το Πρόβλημα 885
Prj06.2 Οι Κλάσεις 887
Prj06.3 Οι Κλάσεις *Course* και *OfferedCourse* 887
Prj06.3.1 Μετατροπες από *Course* σε *OfferedCourse* 890
Prj06.3.2 Οι Ορισμοί των Κλάσεων 890
Prj06.4 Η Κλάση *CourseCollection* 892
Prj06.5 Οι Κλάσεις *Student* και *EnrolledStudent* 899
Prj06.6 Η Κλάση *StudentCollection* 905
Prj06.7 *StudentInCourse* και *StudentInCourseCollection* 909
Prj06.8 Το 1ο Πρόγραμμα (Δημιουργία Αρχείου) 910
Prj06.9 Το 2ο Πρόγραμμα (Χρήση Αρχείου) 912
Prj06.10 Τι Είδαμε σε Αυτό το Παράδειγμα 914

Prj06.1 Το Πρόβλημα

Στην §Prj04.12 παρατηρούσαμε:

- *«Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:*

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που
                                     // δήλωσε
   Course::CourseKey* sCourses;
```

- *Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:*

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
```

```
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)»

Το πρόβλημα που έχουμε να λύσουμε είναι αυτό που λύσαμε στο Project 4 με την εξής διαφορά:

Παίρνοντας υπόψη μας τις παραπάνω παρατηρήσεις «σπάζουμε» την κάθε μια από τις παραπάνω κλάσεις σε δύο ως εξής:

```
class Student
{
public:
// . . .
private:
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
}; // Student

class EnrolledStudent :  public Student
{
public:
// . . .
private:
   char               esSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       esWH;           // ώρες ανά εβδομάδα
   unsigned int       esNoOfCourses;  // αριθμός μαθημάτων που
                                      // δήλωσε
   Course::CourseKey* esCourses;
}; // EnrolledStudent
```

και

```
class Course
{
public:
// . . .
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
}; // Course

class OfferedCourse :  public Course
{
public:
// . . .
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

Στον πίνακα μαθημάτων θα κάνουμε το εξής: αν σε κάποιο μάθημα δεν έγγραφεί ούτε ένας φοιτητής στον πίνακα θα εμφανίζεται με αντικείμενο Course ενώ αν έχει γραφεί έστω και ένας φοιτητής θα έχουμε αντικείμενο κλάσης OfferedCourse. Παρομοίως, στο μητρώο φοιτητών ένας φοιτητής που έχει εγγραφεί σε ένα τουλάχιστον μάθημα εμφανίζεται με αντικείμενο EnrolledStudent· αλλιώς εμφανίζεται με αντικείμενο κλάσης Student.

Παρατήρηση: ▶

«*"Σπάζουμε" την κάθε μια από τις παραπάνω κλάσεις σε δύο*»; Όχι δα! *Τα αντικείμενα κλάσης EnrolledStudent έχουν όλο το περιεχόμενο των αντικειμένων της «παλιάς» κλάσης Student.* Παρομοίως, *τα αντικείμενα κλάσης OfferedCourse έχουν όλο το περιεχόμενο των αντικειμένων της «παλιάς» κλάσης Course.* Προσπαθώντας –με τις νέες κλάσεις– να αντιμετωπίσουμε το πρόβλημα που επισημάναμε καταλήξαμε στα ίδια!

Οι *EnrolledStudent* και *OfferedCourse* θα έλυναν τα πρόβλημα αν *δεν κληρονομούσαν* τις *Student* και *Course* (αλλά περιείχαν τα κλειδιά –*sIdNum* και *cCode*– για αντιστοίχιση.) Ξαναδιάβασε την §23.14.

Το παράδειγμα αυτό έχει στόχο να δείξει άλλα πράγματα και όχι τη σωστή σχεδίαση.◀

Prj06.2 Οι Κλάσεις

Να δούμε πρώτα τις *Course* και *OfferedCourse*. Η κληρονομιά δεν είναι λάθος αφού πέρα από την προγραμματιστική τους σχέση και νοηματικώς λέμε:

Ένα *προσφερόμενο μάθημα* είναι ένα *μάθημα*

Να δούμε δύο ερωτήματα που μπορεί να προκύψουν για πολλούς:

- *Μήπως θα έπρεπε να γράψουμε:* `struct Course` *και να αφήσουμε όλα τα μέλη ανοικτά;* Η αναλλοίωτη της κλάσης δεν μας αφήνει περιθώρια για κάτι τέτοιο.
- *Και θα κρατήσουμε το "*`private`*"; Μήπως πρέπει να το κάνουμε "*`protected`*";* Αν και όταν χρειαστεί θα το κάνουμε· προς το παρόν το κρατούμε όπως είναι.

Παρόμοια σκεπτικά ισχύουν και γαι τις *Student* και *EnrolledStudent*.

Να δούμε τώρα τις συλλογές και πρώτα την *CourseCollection*. Πώς θα αποθηκεύσουμε τον «πίνακα μαθημάτων»; Ας εξετάσουμε δύο περιπτώσεις:

- Σε δύο (δυναμικούς) πίνακες: έναν με στοιχεία τύπου *Course* –στον οποίον αρχικώς φορτώνουμε όλα τα στοιχεία που παίρνουμε από το αρχείο– και έναν με στοιχεία τύπου *OfferedCourse*. Κάθε φορά που βρίσκουμε ένα μάθημα που το δήλωσε κάποιος φοιτητής το μεταφέρουμε από τον πρώτο στον δεύτερο.
- Σε έναν (δυναμικό) πίνακα, τον `ccArr`, αλλά με στοιχεία τύπου *Course*`*` και όχι *Course*. Αρχικώς το κάθε `ccArr[k]` δείχνει ένα (δυναμικό) αντικείμενο κλάσης *Course*, που είναι τα στοιχεία ενός μαθήματος όπως φορτώνονται από το αρχείο. Την πρώτη φορά που βρίσκουμε δήλωση ενός μαθήματος από κάποιον φοιτητή μετατρέπουμε το αντίστοιχο `*ccArr[k]` σε αντικείμενο κλάσης *OfferedCourse*.

Η πρώτη λύση είναι μάλλον πιο απλή. Θα προτιμήσουμε όμως τη δεύτερη για διδακτικούς λόγους.

Για τους ίδιους λόγους θα προτιμήσουμε παρόμοια λύση και για τη *StudentCollection*.

Prj06.3 Οι Κλάσεις *Course* και *OfferedCourse*

Στην κλάση *OfferedCourse* δηλώνεται το μέλος *ocNoOfStudents*. Έτσι, θα πρέπει να μεταφέρουμε εκεί τις μεθόδους *clearStudents()*, *add1Student()*, *delete1Student()* που χειρίζονται το μέλος αυτό. Οι μέθοδοι που χειρίζονται τα υπόλοιπα μέλη δηλώνονται στην *Course* (και κληρονομούνται από την *OfferedCourse*.)

Οι μέθοδοι *save()*, *load()* και *display()* έχουν να κάνουν με ολόκληρο το αντικείμενο είτε είναι τύπου *Course* είτε τύπου *OfferedCourse*. Αυτές θα δηλωθούν "`virtual`" στην *Course* και θα ξαναδηλωθουν στην *OfferedCourse*.

Τέλος, κάθε κλάση θα έχει τους δημιουργούς και τον καταστροφέα της. Και ξεκινούμε από αυτούς.

Ο ερήμην δημιουργός της *Course* χρειάζεται μια μόνον αλλαγή: τη διαγραφή της γραμμής

```
cNoOfStudents = 0;
```

ενώ ο καταστροφέας θα γίνει:

```
virtual ~Course() { };
```

Ας δούμε τώρα τα αντίστοιχα για την *OfferedCourse*. Θα γράψουμε τον δημιουργό όπως μάθαμε στην §23.3:

```
OfferedCourse::OfferedCourse( string aCode, string aTitle )
   : Course( aCode, aTitle ), ocNoOfStudents( 0 )  { }
```

Όπως βλέπεις, τα πάντα γίνονται με τη λίστα εκκίνησης και το σώμα της συνάρτησης είναι κενό.

Ο καταστροφέας είναι τετριμμένος:

```
virtual ~OfferedCourse() { };
```

Να έλθουμε τώρα στις τρεις μεθόδους. Αυτές δηλώνονται στην *Course* ως εξής:

```
virtual void save( ostream& bout ) const;
virtual void load( istream& bin );
virtual void display( ostream& tout ) const;
```

Στους ορισμούς τους θα γίνουν οι εξής αλλαγές:

- Στη *save(0* διαγράφουμε την εντολή:

```
bout.write( reinterpret_cast<const char*>(&cNoOfStudents),
            sizeof(cNoOfStudents) );
```

- Στη *load()* διαγράφουμε την εντολή:

```
bin.read( reinterpret_cast<char*>(&tmp.cNoOfStudents),
          sizeof(cNoOfStudents) ); // αριθ. φοιτ. στο μάθημα
```

- Στη *display()* διαγράφουμε το: "`<< '\t' << cNoOfStudents`"

Στην *OfferedCourse* κάνουμε την ίδια δήλωση, παρ' όλο που τα "`virtual`" δεν είναι απαραίτητα. Οι ορισμοί θα είναι:

```
void OfferedCourse::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw CourseXptn( CourseKey(getCode()), "save",
                        CourseXptn::fileNotOpen );
   this->Course::save( bout );
   bout.write( reinterpret_cast<const char*>(&ocNoOfStudents),
               sizeof(ocNoOfStudents) ); // αριθ. φοιτ. στο μάθημα
   if ( bout.fail() )
      throw CourseXptn( CourseKey(getCode()), "save",
                        CourseXptn::cannotWrite );
} // OfferedCourse::save
```

Όπως βλέπεις, με τη "`this->Course::save(bout)`" φυλάγουμε το υποαντικείμενο κλάσης *Course* και στη συνέχεια φυλάγουμε και την τιμή του μέλους *ocNoOfStudents*.

```
void OfferedCourse::load( istream& bin )
{
   OfferedCourse tmp;
   tmp.Course::load( bin );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&tmp.ocNoOfStudents),
                sizeof(ocNoOfStudents) );// αριθ. φοιτ. στο μάθημα
```

```
      if ( bin.fail() )
         throw CourseXptn( CourseKey(getCode()), "load",
                           CourseXptn::cannotRead);
   } // if ( !bin.eof() ). . .
   *this = tmp;
} // OfferedCourse::load
```

Και εδώ, με την "`tmp.Course::load(bin)`" φορτώνουμε το υποαντικείμενο κλάσης *Course* του *tmp*. Στη συνέχεια φορτώνουμε την τιμή του μέλους *tmp*`.`*ocNoOfStudents*.

```
void OfferedCourse::display( ostream& tout ) const
{
   this->Course::display( tout );
   tout << ocNoOfStudents << endl;
   if ( tout.fail() )
      throw CourseXptn( CourseKey(getCode()), "display",
                        CourseXptn::cannotWrite );
} // OfferedCourse::display
```

Σε όλες τις ρίψεις εξαιρέσεων πρόσεξε τα εξής:

- Συνεχίζουμε να ρίχνουμε εξαιρέσεις *CourseXptn* αφού τα προβλήματα που βρίσκουμε τα έχουμε προβλέψει και για τη βασική κλάση και δεν παραπλανούμε αυτόν που θα συλλάβει την εξαίρεση. Στη συνέχεια πάντως θα ορίσουμε και μια *OfferedCourseXptn*.
- Αντικαταστήσαμε το "`cCode`" με το "`CourseKey(getCode())`". Αυτό είναι απαραίτητο διότι τα (`private`) μέλη της βασικής δεν είναι ορατά από την παράγωγη. Να το βάλουμε "`protected`"; Όχι αφού το χρησιμοποιούμε μόνο στη ρίψη εξαίρεσης.

Οι τρεις μέθοδοι διαχείρισης του μέλους *ocNoOfStudents* είναι: οι δύο **inline**

```
   void clearStudents() {  ocNoOfStudents = 0;  }
   void add1Student() {  ++ocNoOfStudents;  }
```

και η:

```
void OfferedCourse::delete1Student()
{
   if ( ocNoOfStudents <= 0 )
      throw OfferedCourseXptn( CourseKey(getCode()),"delete1Student",
                               OfferedCourseXptn::noStudent );
   --ocNoOfStudents;
} // OfferedCourse::delete1Student
```

Εδώ, όπως βλέπεις, χρησιμοποιούμε άλλη κλάση εξαιρέσεων διότι το πρόβλημα που εμφανίζεται δεν έχει οποιαδήποτε σχέση με τη βασική κλάση (όπως τα προβλήματα με τα ρεύματα.)

Όπως είπαμε (§23.7), «*η κλάση εξαιρέσεων της παράγωγης κλάσης είναι παράγωγη της κλάσης εξαιρέσεων της βασικής κλάσης.*» Έχουμε λοιπόν:

```
struct OfferedCourseXptn :  public CourseXptn
{
   enum { noStudent=20 };
   OfferedCourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
                      const char* sv="" )
      : CourseXptn( obk, mn, 0, sv ) {  errorCode = ec; }
}; // OfferedCourseXptn
```

Στην περίπτωσή μας, όπως βλέπεις, η παράγωγη κλάση διαφέρει από τη βασική μόνον στον ορισμό της σταθεράς *noStudent*. Γιατί βάλαμε εκείνο το "`=20`"; Διότι αλλιώς το *OfferedCourseXptn*`::`*noStudent* θα είχε τιμή "`0`" όπως ακριβώς και το *OfferedCourseXptn*`::`*keyLen* (κληρονομιά!)

Στο σώμα του δημιουργού το μόνο που κάνουμε είναι να δώσουμε τον σωστό κωδικό λάθους.

Prj06.3.1.1 Μετατροπες από *Course* σε *OfferedCourse*

Πρόσεξε τώρα ένα πρόβλημα που θα αντιμετωπίσουμε: Όπως λέγαμε «*Την πρώτη φορά που βρίσκουμε δήλωση ενός μαθήματος από κάποιον φοιτητή μετατρέπουμε το αντίστοιχο* `*ccArr[k]` *σε αντικείμενο κλάσης OfferedCourse.*» Πώς θα γίνεται αυτό; Ας πούμε ότι έχουμε δηλώσει:

```
   OfferedCourse* pOneCourse( new OfferedCourse );
```

Στο αντικείμενο που δείχνει το *pOneCourse* θα πρέπει να αντιγράψουμε το περιεχόμενο του `*ccArr[ndx]`, να μηδενίσουμε το *ocNoOfStudents* και μετά να βάλουμε:

```
      delete ccArr[ndx];
      ccArr[ndx] = pOneCourse;
```

Πώς μπορούμε να αντιγράψουμε το περιεχόμενο;

- Γράφοντας τον κατάλληλο δημιουργό μετατροπής» που από ένα αντικείμενο της κλάσης *Course* θα δημιουργεί αντικείμενο κλάσης *OfferedCourse* με τιμή "`0`" στο *ocNoOfStudents*. Έτσι θα μπορούμε να γράψουμε:

```
      OfferedCourse* pOneCourse( new OfferedCourse(*ccArr[ndx]) );
```

- Επιφορτώνοντας τον τελεστή εκχώρησης ώστε να κάνουμε νόμιμη την εντολή:

```
   *pOneCourse = *ccArr[ndx];
```

Προφανώς η πρώτη επιλογή είναι ταχύτερη αφού δημιουργεί κατ' ευθείαν το αντικείμενο που μας ενδιαφέρει. Δηλώνουμε λοιπόν:

```
   OfferedCourse( const Course& oneCourse );
```

και ορίζουμε, όπως κάναμε και στον ερήμην δημιουργό:

```
OfferedCourse::OfferedCourse( const Course& oneCourse )
   : Course( oneCourse ), ocNoOfStudents( 0 )  { }
```

Παρατηρήσεις: ▶

1. Δεν επιφορτώνουμε και τον τελεστή εκχώρησης για να δοκιμάσουμε και την άλλη επιλογή; Δεν χρειάζεται! Αφού έχουμε σωστό τελεστή εκχώρησης και τον δημιουργό "`OfferedCourse( const Course& oneCourse )`" έχουμε «νομιμοποιήσει» και την εκώρηση "`*pOneCourse = *ccArr[ndx]`".
2. Δεν θα χρειαστούμε και έναν δημιουργό

```
   Course( const OfferedCourse& oneOfrdCourse );
```

για να κάνουμε την αντίστροφη μετατροπή; Όχι· αυτό γίνεται –με τεμαχισμό φυσικά– από τον (συναγόμενο) δημιουργό αντιγραφής της *Course*.◀

Prj06.3.1.2 Οι Ορισμοί των Κλάσεων

Τώρα η *Course* θα είναι:[1]

```
class Course
{
public:
   enum { cCodeSz = 8 };
   struct CourseKey
   {
      char s[cCodeSz];
      explicit CourseKey( string aKey="" )
      {  strncpy( s, aKey.c_str(), cCodeSz-1 );
         s[cCodeSz-1] = '\0';  }
   }; // CourseKey
   explicit Course( string aCode="", string aTitle="" );
   virtual ~Course() { };
// getters
```

[1] Αυτή δεν είναι άλλη από τη *SylCourse* που λέγαμε στο Project 2· φυσικά είναι πιο καλογραμμένη (πέρα από την προετοιμασία για την κληρονομιά.)

```
   const char* getCode() const {  return cCode.s;  }
   const char* getTitle() const {  return cTitle;  }
   unsigned int getFSem() const {  return cFSem;  }
   bool getCompuls() const {  return cCompuls;  }
   char getSector() const {  return cSector;  }
   const char* geCateg() const {  return cCateg;  }
   unsigned int getWH() const {  return cWH;  }
   unsigned int getUnits() const {  return cUnits;  }
   const char* getPrereq() const {  return cPrereq.s;  }
// setters
   void setCode( string aCode );
   void setTitle( string aTitle );
   void setFSem( int aFSem );
   void setCompuls( bool aCompuls ) {  cCompuls = aCompuls;  };
   void setSector( char aSector );
   void setCateg( string aCateg );
   void setWH( int aWH );
   void setUnits( int aUnits );
   void setPrereq( const string& prCode );
// other
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout ) const;
private:
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
}; // Course

typedef Course*  PCourse;
```

Οι επιφορτώσεις των "`!=`" και "`==`" για τους τύπους *CourseKey* και *Course* δεν αλλάζουν.

Η κλάση εξαιρέσεων είναι:

```
struct CourseXptn
{
   enum { keyLen, rangeError, noSuchSector, noSuchCateg, autoRef,
          fileNotOpen, cannotWrite, cannotRead };
   Course::CourseKey objKey;
   char              funcName[100];
   int               errorCode;
   char              errStrVal[100];
   int               errIntVal;
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, const char* sv="" )
      : objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   CourseXptn( const Course::CourseKey& obk, const char* mn,
               int ec, int iv )
      : objKey( obk ),  errorCode( ec ),  errIntVal( iv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // CourseXptn
```

Η μόνη διαφορά από την προηγούμενη μορφή: έχει αφαιρεθεί ο ορισμός της σταθεράς *noStudent*.

Η παράγωγη κλάση:

```
class OfferedCourse :  public Course
{
public:
   explicit OfferedCourse( string aCode="", string aTitle="" );
```

```
   OfferedCourse( const Course& oneCourse );
   virtual ~OfferedCourse() { };
// getters
   unsigned int getNoOfStudents() const { return ocNoOfStudents; }
// setters
   void clearStudents() {  ocNoOfStudents = 0;  }
   void add1Student() {  ++ocNoOfStudents;  }
   void delete1Student();
// other
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout ) const;
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse

typedef OfferedCourse*  POfferedCourse;
```

Πρόσεξε ότι δεν μας χρειάζονται επιφορτώσεις για συγκρίσεις αντικειμένων κλάσης *OfferedCourse*: Μας αρκούν αυτές που έχουμε για την κλάση *Course*.

Τώρα βέβαια, το «*δεν μας χρειάζονται*» είναι «μεγάλη κουβέντα» διότι με τις επιφορτώσεις των τελεστών της βασικής κλάσης μπορούμε να συγκρίνουμε και αντικείμενα των δύο κλάσεων και να βρούμε ότι ένα αντικείμενο κλάσης *Course* είναι ίσο με ένα αντικείμενο κλάσης *OfferedCourse* επειδή έχουν τον ίδιο κωδικό! Είναι σωστό αυτό; Ας μην ανοίξουμε τώρα φιλοσοφική συζήτηση πάνω στο θέμα·[2] να πούμε μόνο το εξής: Αυτές οι συγκρίσεις μας χρειάζονται για την αναζήτηση μέσα στη συλλογή. Για τη δουλειά αυτή είναι σωστές!

Η κλάση εξαιρέσεων είναι αυτή που είδαμε και πιο πάνω:

```
struct OfferedCourseXptn :  public CourseXptn
{
   enum { noStudent=20 };
   OfferedCourseXptn( const Course::CourseKey& obk, const char* mn, int ec,
                      const char* sv="" )
      : CourseXptn( obk, mn, 0, sv ) {  errorCode = ec; }
}; // OfferedCourseXptn
```

Prj06.4 Η Κλάση *CourseCollection*

Η βασική αλλαγή στην κλάση *CourseCollection* είναι στη δήλωση του δυναμικού πίνακα που από

```
   Course*   ccArr;
```

γίνεται πίνακας βελών:

```
   PCourse*  ccArr;
```

Επειδή, όπως πρέπει ήδη να κατάλαβες, η σπουδαιότερη μέθοδος στις κλάσεις-συλλογές είναι η *findNdx()*, θα πρέπει να ξεκινήσουμε από αυτήν. Η μεθοδος βασίζεται στη *linSearch()* από τη βιβλιοθήκη περιγραμμάτων **MyTmpltLib**. Η *linSearch()* δεν μπορεί να δουλέψει για τον νέο πίνακα διότι δεν μπορούμε να επιφορτώσουμε τελεστές για τύπους βελών. Το ίδιο πρόβλημα θα έχουμε και στη *findNdx()* της *StudentCollection*. Τι μπορούμε να κάνουμε για να λύσουμε αυτό το πρόβλημα;

- Να γράψουμε μέσα στις *findNdx()* τις εντολές αναζήτησης και να μη χρειαζόμαστε τη *linSearch()*.
- Να «κρύψουμε» σε μια κλάση το βέλος ώστε να μπορούμε να επιφορτώσουμε τους τελεστές σύγκρισης για την κλάση αυτή.
- Να γράψουμε μια νέα εκδοχή της *linSearch()* –ας την πούμε *linSearchP*– για πίνακες βελών.

[2] Πάντως δεν είναι κακό να το σκεφτείς λιγάκι και να το συζητήσεις με συναδέλφους σου...

Η τελευταία επιλογή φαίνεται πιο κομψή (και πιο απλή) από τις άλλες. Γράφουμε λοιπόν:

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

Όπως βλέπεις, αυτό που συγκρίνεται με τον "`!=`" είναι το "`*v[k]`".

Βλέπεις όμως και κάτι άλλο: πρέπει να υπάρχει το "`*v[upto+1]`". Αν ψάχνουμε σε ολόκληρον τον πίνακα αυτό σημαίνει ότι θα πρέπει να υπάρχει το "`*v[n]`" (το στοιχείο που κρατούμε για φρουρό στις αναζητήσεις).

Τώρα, μπορούμε να γράψουμε την

```
int CourseCollection::findNdx( const string& code ) const
{
   int ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = -1;
   else
      ndx = linSearchP( ccArr, ccNOfCourses,
                        0, ccNOfCourses-1, Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

που, όπως βλέπεις, δεν διαφέρει από την αρχική, παρά μόνο στο όνομα της συνάρτησης αναζήτησης.

Θα συνεχίσουμε βλέποντας τις επιπτώσεις των αλλαγών που κάναμε στον ερήμην δημιουργό:

```
CourseCollection::CourseCollection()
{
   try
   {
      ccReserved = ccIncr;
      ccArr = new PCourse[ ccReserved ];
      ccNOfCourses = 0;
      ccPAllEnrollments = 0; // NULL
      ccArr[ccNOfCourses] = new Course;
      for ( int k(ccNOfCourses+1); k < ccReserved; ++k )
         ccArr[k] = 0;
   }
   catch( bad_alloc& )
   {
      throw CourseCollectionXptn( "CourseCollection",
                                  CourseCollectionXptn::allocFailed);
   }
} // CourseCollection::CourseCollection
```

Πρόσεξε την εντολή "`ccArr[ccNOfCourses] = new Course`" (το στοιχείο για τον φρουρό). Η **for** που ακολουθεί διασφαλίζει το ότι τα βέλη που δεν χρησιμοποιούνται έχουν τιμή "`0`".

Φυσικά αλλάζει και ο καταστροφέας:

```
CourseCollection::~CourseCollection()
{
   for ( int k(0); k <= ccNOfCourses; ++k )
      delete ccArr[k];
   delete[] ccArr;
} // CourseCollection::~CourseCollection
```

Παρόμοιες επιπτώσεις έχουμε και στις μεθόδους (**private**) «χαμηλού επιπέδου» *erase1Course()* και *insert1Course()*.

```
void CourseCollection::erase1Course( int ndx )
{
   delete ccArr[ndx];
   ccArr[ndx] = ccArr[ccNOfCourses-1];
   ccArr[ccNOfCourses-1] = ccArr[ccNOfCourses];
   ccArr[ccNOfCourses] = 0;
   --ccNOfCourses;
} // CourseCollection::erase1Course
```

Η "`ccArr[ndx] = ccArr[ccNOfCourses-1]`" δεν αρκεί πια:

- Πριν από αυτήν πρέπει να ανακυκλώσουμε το "`*ccArr[ndx]`".
- Μετά από αυτήν πρέπει να βάλουμε τον φρουρό στη σωστή θέση. Αυτό γίνεται με την "`ccArr[ccNOfCourses-1] = ccArr[ccNOfCourses]`".

Με την *insert1Course()* έχουμε πρόβλημα· δες την επικεφαλίδα της:

```
void CourseCollection::insert1Course( const Course& aCourse );
```

Αυτή μας επιτρέπει να εισαγάγουμε στη συλλογή ένα αντικείμενο *Course* αλλά και ένα αντικείμενο *OfferedCourse*. Στη πρώτη περίπτωση θα δώσουμε

```
ccArr[ccNOfCourses] = new Course( aCourse );
```

ενώ στη δεύτερη

```
ccArr[ccNOfCourses] = new OfferedCourse( aCourse );
```

Τι θα πρέπει να κάνουμε;

- Μια λύση είναι η *τυποθεώρηση της παραμέτρου αναφοράς* (§23.13.1). Αυτή έχει –στην περίπτωσή μας– το εξής πλεονέκτημα: Δεν θα χρειαστεί να αλλάξουμε την *add1Course()*.
- Μια άλλη λύση είναι η εξής: αλλάζουμε την παράμετρο στην επικεφαλίδα:

```
void CourseCollection::insert1Course( const Course* pCourse )
```

Με δυναμική τυποθεώρηση μπορούμε να βρούμε τον τύπο του αντικειμενου που δείχνει το *pCourse*. Μειονέκτημα: μπορεί να κληθεί με όρισμα τη διεύθυνση μη-δυναμικού αντικειμένου. Αυτό όμως μπορεί να ελεγχθεί αφού η *insert1Course()* είναι **private** και καλείται μόνον από τις (*CourseCollection*::) *add1Course()* και (*CourseCollection*::)*load()*. Αυτό που φαίνεται να είναι πλεονέκτημα είναι η απλότητά της· αλλά η αλήθεια είναι ότι η τυποθεώρηση της παραμέτρου αναφοράς μεταφέρεται στην *add1Course()*.

Προτιμούμε λοιπόν τη δεύτερη λύση.

```
void CourseCollection::insert1Course( Course* pCourse )
{
   if ( ccReserved <= ccNOfCourses+1 )
   {
      try
      {
         renew( ccArr, ccNOfCourses+1, ccReserved+ccIncr );
         ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses];  // αλλαγή φρουρού
         for ( int k(ccNOfCourses+2); k < ccReserved+ccIncr; ++k )
            ccArr[k] = 0;
```

```
         ccReserved += ccIncr;
      }
      catch( ... )
      {  throw CourseCollectionXptn( "insert1Course",
                                     CourseCollectionXptn::allocFailed );  }
   }
   ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses];
   ccArr[ccNOfCourses] = pCourse;
   ++ccNOfCourses;
} // CourseCollection::insert1Course
```

Όσο για τις άλλες αλλαγές:

- Κρατούμε την πρώτη από τις «νέες θέσεις» (*ccNOfCourses*+1) στον πίνακα για τον φρουρό ("`ccArr[ccNOfCourses+1] = ccArr[ccNOfCourses]`"). Στις υπόλοιπες βάζουμε "`0`".
- Εισάγουμε τη νέα τιμή στον «παλιό φρουρό» ("`ccArr[ccNOfCourses] = pCourse`").
- Κάνουμε (με αρκετή αυθαιρεσία) την υπόθεση: αν κάτι δεν πάει καλά θα πει ότι έχουμε πρόβλημα δυναμικής μνήμης. Έτσι όποιουδήποτε τύπου ("`catch(...)`") εξαίρεση και να πιάσουμε εμείς ρίχνουμε *allocFailed*!

Στις αντίστοιχες μεθόδους (**public**) «υψηλού επιπέδου» η *delete1Course()* θα γίνει:

```
void CourseCollection::delete1Course( string code )
{
   int ndx( findNdx(code) );
   if ( ndx >= 0 )  // υπάρχει
   {
      *ccArr[ccNOfCourses] = Course();
      ccArr[ccNOfCourses]->setPrereq( code ); // φρουρός
      int k(0);
      while ( strcmp(ccArr[k]->getPrereq(), code.c_str()) != 0 )
         ++k;
      if ( k < ccNOfCourses )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::prereqRef,
                                     code.c_str() );
      OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
      if ( pOneCourse != 0 )
      {
         int enrStdnt( pOneCourse->getNoOfStudents() );
         if ( enrStdnt > 0 ) // υπάρχουν εγγεγραμμένοι φοιτητές
            throw CourseCollectionXptn( "delete1Course",
                                        CourseCollectionXptn::enrollRef,
                                        code.c_str(), enrStdnt );
      }
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Πέρα από τις αλλαγές από "`.`" σε "`->`" πρόσεξε αυτά που κάνουμε στο τέλος της: Με δυναμική τυποθεώρηση ελέγχουμε αν το `*ccArr[ndx]` είναι τύπου *OfferedCourse*.

- Αν δεν είναι (*pOneCourse* == 0) προχωρούμε κατ' ευθείαν στην "`erase1Course(ndx)`".
- Αν είναι (*pOneCourse* != 0) κάνουμε επιπλέον τον έλεγχο

  pOneCourse->*getNoOfStudents()* > 0

  Προχωρούμε στην "`erase1Course(ndx)`" μόνον αν δεν ισχύει αυτή η συνθήκη.

Η επικεφαλίδα της *add1Course()* είναι:

```
void CourseCollection::add1Course( const Course& aCourse )
```

Γιατί να μην αλλάξουμε την παράμετρο σε "`const Course* pCourse`" και να κάνουμε τη ζωή μας πιο εύκολη; Διότι –αφού είναι **public**– μπορεί να υπάρχουν εφαρμογές που τη χρησιμοποιούν ήδη και θα πρέπει να πάμε να κάνουμε αλλαγές και στις εφαρμογές αυτές.

Όπως καταλαβαίνεις, το πρόβλημα που είδαμε –και παρακάμψαμε– παραπάνω ξαναεμφανίζεται και πρέπει να λυθεί τώρα.

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   int ndx( findNdx(aCourse.getCode()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )    // υπάρχει προαπαιτούμενο
      {
         int ndx( findNdx(aCourse.getPrereq()) );
         if ( ndx < 0 )           // δεν βρέθηκε το κλειδί του προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::prereqRef,
                                        aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      try
      {
         try
         {
            const OfferedCourse&
                     oneCourse( dynamic_cast<const OfferedCourse&>(aCourse) );
            insert1Course( new OfferedCourse(oneCourse) );
         }
         catch( bad_cast )
         {  insert1Course( new Course(aCourse) );  }
      }
      catch( bad_alloc )
      {  throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::allocFailed );  }
   }
} // CourseCollection::add1Course
```

Πρόσεξε τι κάνουμε: αφού σιγουρευτούμε ότι πρέπει να κάνουμε την εισαγωγή δηλώνουμε

```
            const OfferedCourse&
                     oneCourse( dynamic_cast<const OfferedCourse&>(aCourse) );
```

Αν η πραγματική παράμετρος που έρχεται στην *aCourse* είναι κλάσης *OfferedCourse* τότε η τιμή της θα αντιγραφεί ως αρχική τιμή της *oneCourse*. Αλλιώς, αν είναι κλάσης *Course*, ρίχνεται εξαίρεση *bad_cast*. Αυτά γίνονται στην εσωτερική **try/catch** που, όπως βλέπεις, δεν χειρίζεται κάποια «εξαιρετική» κατάσταση αλλά λειτουργεί σαν **ifelse**. Αυτή είναι μια ανορθόδοξη χρήση του μηχανισμού εξαιρέσεων.

Σε κάθε περίπτωση, θα εκτελεσθεί μια **new**· η εξωτερική **try/catch** υπάρχει για τυχόν απότυχία της.

Φυσικά, η *clearStudents()* καλείται μόνον στην περίπτωση που το αντικείμενο που εισάγεται είναι κλάσης *OfferedCourse*.

Ενδιαφέρον έχουν οι μέθοδοι πρόσθεσης φοιτητή σε μάθημα και διαγραφής φοιτητή από μάθημα· *σε αυτές θα έχουμε και τις αλλαγές τύπου*.

```
void CourseCollection::add1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
   if ( pOneCourse == 0 )
   {
      try {  pOneCourse  = new OfferedCourse( *ccArr[ndx] );  }
```

```
      catch( bad_alloc& )
      {
         throw CourseCollectionXptn( "add1Student",
                                     CourseCollectionXptn::allocFailed,
                                     code.c_str() );
      }
      delete ccArr[ndx];
      ccArr[ndx] = pOneCourse;
   }
   pOneCourse->add1Student();
} // CourseCollection::add1Student
```

Κατ' αρχάς να τονίσουμε ότι δεν μπορείς να γράψεις "`ccArr[ndx]->add1Student()`": ο μεταγλωττιστής θα σου πει ότι η κλάση *Course* δεν έχει μέθοδο *add1Student()*. Έτσι, παίρνουμε ένα νέο βέλος, το *pOneCourse*, που δείχνει το ίδιο αντικείμενο με το `ccArr[ndx]`. Αν όμως το `*ccArr[ndx]` είναι τύπου *Course* η μετατροπή θα αποτύχει. Στην περίπτωση αυτήν κάνουμε τη μετατροπή όπως είπαμε στην §Prj06.3.1. Σε κάθε περίπτωση, όταν έρχεται η ώρα να εκτελεσθεί η "`pOneCourse->add1Student()`" το *pOneCourse* δείχνει το ίδιο αντικείμενο (τύπου *OfferedCourse*) με το `ccArr[ndx]`.

```
void CourseCollection::delete1Student( string code )
{
   int ndx( findNdx(code) );
   if ( ndx < 0 )
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[ndx]) );
   pOneCourse->delete1Student();
// τα παρακάτω μπορεί και να περισσεύουν . . .
   if ( pOneCourse->getNoOfStudents() == 0 )
   {
      Course* tmp( new Course(*pOneCourse) );
      delete pOneCourse;
      ccArr[ndx] = tmp;
   }
} // CourseCollection::delete1Student
```

Τι καινούριο υπάρχει εδώ; Αν μετά τη διαγραφή ενός φοιτητή μηδενίζεται το πλήθος των φοιτητών που θέλουν το μάθημα μετατρέπουμε το αντικείμενο που δείχνει το `ccArr[ndx]` από *OfferedCourse* σε *Course*.

Και είναι απαραίτητη αυτή η μετατροπή; Μετά από λίγο μπορεί να έλθει απαίτηση άλλου φοιτητή να εγγραφεί στο μάθημα. Σωστό! Το τι θα κάνουμε εξαρτάται από την εφαρμογή. Αν έχεις ένα πρόγραμμα που θέλει γρήγορες αποκρίσεις

- Δεν κάνεις αυτήν τη μετατροπή.
- Πριν από τη φύλαξη των στοιχείων της συλλογής σαρώνεις όλα τα μαθήματα και όπου βρίσκεις *OfferedCourse* με *ocNoOfStudents* `==` 0 το αλλάζεις σε *Course*.

Στην εφαρμογή του παραδείγματος «δεν μας κυνηγάει κανείς»…

Ας έλθουμε τώρα στις *save()* και *load()*. Το πρόβλημά μας είναι ότι στο ίδιο μη-μορφοποιημένο αρχείο θα έχουμε αντικείμενα δύο διαφορετικών κλάσεων: *Course* και *OfferedCourse*. Πώς το λύνουμε; Με ευρετήριο! Αλλά τώρα, εκτός από το κλειδί και τη θέση, θα έχουμε ένδειξη και για την κλάση του κάθε αντικειμενου. Για παράδειγμα:

```
struct CIndexEntry
{
   Course::CourseKey cCode;
   size_t            loc;
   short int         inhLvl;
}; // IndexEntry
```

που σημαίνει: το αντικείμενο με κλειδί (κωδικό) *cCode* αρχίζει από τη θέση *loc* του αρχείου και είναι κλάσης *Course* αν το *inhLvl* έχει τιμή "`0`" ή κλάσης *OfferedCourse* αν έχει τιμή "`1`".

Για τις αναζητήσεις θα χρειαστούμε έναν δημιουργό –που να δημιουργεί αντικείμενο από τον κωδικό– και επιφόρτωση του "!=":

```
struct CIndexEntry
{
   Course::CourseKey cCode;
   size_t            loc;
   short int         inhLvl;
   explicit CIndexEntry( Course::CourseKey aCode=Course::CourseKey(""),
                         int aLoc=0, short int aInhL=0 )
      :  cCode( aCode ),  loc( aLoc ),  inhLvl( aInhL )  {  }
}; // CIndexEntry

bool operator!=( const CIndexEntry& lhs, const CIndexEntry& rhs )
{  return ( lhs.cCode != rhs.cCode );  }

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

Έτσι λοιπόν γράφουμε:

```
void CourseCollection::save( ofstream& bout, CIndexEntry* crsIndex ) const
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( int k(0); k < ccNOfCourses; ++k )
   {
      crsIndex[k].cCode = Course::CourseKey(ccArr[k]->getCode());
      crsIndex[k].loc = bout.tellp();
      OfferedCourse* pOneCourse( dynamic_cast<OfferedCourse*>(ccArr[k]) );
      if ( pOneCourse == 0 )  // Course
      {  crsIndex[k].inhLvl = 0;
         ccArr[k]->save( bout );  }
      else                    // OfferedCourse
      {  crsIndex[k].inhLvl = 1;
         pOneCourse->save( bout );  }
   }
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::cannotWrite );
} // CourseCollection::save
```

Πρόσεξε ότι και εδώ η διαχείριση της δυναμικής μνήμης όπου υλοποιείται το ευρετήριο θα πρέπει να γίνεται από τη συνάρτηση που καλεί τη *save()*.

Δες τώρα πώς χρησιμοποιούμε το ευρετήριο κατά τη φόρτωση:

```
void CourseCollection::load( ifstream& bin, const CIndexEntry* crsIndex )
{
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(n) );
   if ( !bin.eof() )
   {
      CourseCollection tmp;
      try
      {
         for ( int k(0); k < n && !bin.fail(); ++k )
         {
            Course* pOneCourse;
            if ( crsIndex[k].inhLvl == 0 )  // Course
               pOneCourse = new Course;
            else                            // OfferedCourse
               pOneCourse = new OfferedCourse;
            pOneCourse->load( bin );
            tmp.insert1Course( pOneCourse );
         }
      }
```

```
      catch( bad_alloc& )
      {
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::allocFailed );
      }
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Prj06.5 Οι Κλάσεις *Student* και *EnrolledStudent*

Από την κλάση *Student* αφαιρούνται (και δηλώνονται στην *EnrolledStudent*) τα μέλη που έχουν σχέση με τον δυναμικό πίνακα κωδικών (*sNoOfCourses*, *sCourses*, *sReserved*) και το *sWH* (σύνολο εβδομαδιαίων ωρών διδασκαλίας που προκύπτει από τις δηλώσεις μαθημάτων). Φυσικά, ακολουθούνται από τις μεθόδους χειρισμού αυτών των μελών, δηλαδή: *getNoOfCourses()*, *getCourses()*, *clearCourses()*, *find1Course()*, *add1Course()*, *delete1Course()*, *findNdx()*, *insert1Course()*, *erase1Course()* και *getWH()*.

Τώρα, όλα απλουστεύονται. Ο ερήμην δημιουργός θα είναι:

```
Student::Student( int aIdNum )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum, aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
}; // Student()
```

ενώ δεν χρειάζεται να γράψουμε δημιουργό αντιγραφής. Παρομοίως, δεν χρειάζεται να γράψουμε επιφόρτωση του τελεστή εκχώρησης. Θα γράψουμε όμως έναν (κενό) καταστροφέα:

```
   virtual ~Student() { };
```

Οι παρακάτω μέθοδοι θα υπάρχουν και στην παράγωγη κλάση και για τον λόγο αυτό θα πρέπει να δηλωθούν:

```
   virtual void swap( Student& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
```

και θα είναι αναλόγως απλουστευμένες:

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );
} // Student::swap

void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
```

```
   bout.write( sFirstname, sizeof(sFirstname) );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save

void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load

void Student::display( ostream& tout )
{
   tout << sIdNum << '\t' << sSurname << '\t' << sFirstname << endl
} // Student::display
```

Έτσι, η (βασική) κλάση *Student* θα είναι:

```
class Student
{
public:
// constructors, destructor
   explicit Student( int aIdNum=0 );
   virtual ~Student() { };
// getters
   unsigned int getIdNum() const { return sIdNum; }
   const char* getSurname() const { return sSurname; }
   const char* getFirstname() const { return sFirstname; }
// setters
   void setIdNum( int aIdNum );
   void setSurname( string aSurname );
   void setFirstname( string aFirstname );
// other methods
   void readPartFromText( istream& tin );
   virtual void swap( Student& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
private:
   enum { sNameSz = 20 };
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSurname[sNameSz];
   char               sFirstname[sNameSz];

   unsigned int countTabs( string aLine );
}; // Student

typedef Student* PStudent;
```

Η επιφόρτωση των "`!=`" και "`==`" δεν αλλάζει.

Η κλάση εξαιρέσεων απλουστεύεται κάπως:

```
struct StudentXptn
{
   enum { negIdNum, incomplete, fileNotOpen, cannotRead, cannotWrite };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   StudentXptn( int obk, const char* mn, int ec, const char* sv="" )
```

```
      :  objKey( obk ),  errorCode( ec )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
   StudentXptn( int obk, const char* mn, int ec, int ev )
      :  objKey( obk ),  errorCode( ec ),  errIntVal( ev )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // StudentXptn
```

Τα πιο πολύπλοκα κομμάτια των δημιουργών μεταφέρονται στην παράγωγη κλάση *EnrolledStudent*. Ας αρχίσουμε από τον ερήμην δημιουργό που δηλώνεται ως:

```
   explicit EnrolledStudent( int aIdNum=0 );
```

και ορίζεται:

```
EnrolledStudent::EnrolledStudent( int aIdNum )
   :  Student( aIdNum ),  esWH( 0 )
{
   try  {  esCourses = new Course::CourseKey[ esIncr ];  }
   catch( bad_alloc )
   {  throw StudentXptn( getIdNum(), "EnrolledStudent",
                         EnrolledStudentXptn::allocFailed );  }
   esReserved = esIncr;
   esNoOfCourses = 0;
} // EnrolledStudent::EnrolledStudent
```

Φυσικά, τώρα χρειαζόμαστε και δημιουργό αντιγραφής και επιφόρτωση του αντιγραφικού τελεστή εκχώρησης. Ο δημιουργός αντιγραφής δηλώνεται:

```
   EnrolledStudent( const EnrolledStudent& rhs );
```

και ορίζεται:

```
EnrolledStudent::EnrolledStudent( const EnrolledStudent& rhs )
   :  Student( rhs ),  esWH( rhs.esWH )
{
   try  {  esCourses = new Course::CourseKey[ rhs.esReserved ];  }
   catch( bad_alloc )
   {  throw EnrolledStudentXptn( getIdNum(), "Student",
                                 EnrolledStudentXptn::allocFailed );  }
   esReserved = rhs.esReserved;
   for ( int k(0); k < rhs.esNoOfCourses; ++k )
      esCourses[k] = rhs.esCourses[k];
   esNoOfCourses = rhs.esNoOfCourses;
}; // EnrolledStudent::EnrolledStudent
```

Για να επιφορτώσουμε τον τελεστή εκχώρησης όπως ξέρουμε θα χρειαστούμε τη *swap()* που τη δηλώνουμε:

```
   virtual void swap( EnrolledStudent& rhs );
```

και την ορίζουμε:

```
void EnrolledStudent::swap( EnrolledStudent& rhs )
{
   Student::swap( rhs );

   std::swap( esWH, rhs.esWH );
   std::swap( esNoOfCourses, rhs.esNoOfCourses );

   Course::CourseKey* svck( esCourses );
   esCourses = rhs.esCourses;  rhs.esCourses = svck;

   std::swap( esReserved, rhs.esReserved );
} // EnrolledStudent::swap
```

Όπως βλέπεις, αυτή στηρίζεται στη *swap()* της βασικής κλάσης. Πράγματι, η "`Student::swap(rhs)`" –που μπορείς να τη γράψεις και "`this->Student:: swap(rhs)`"– ανταλλάσσει τις τιμές των υποαντικειμένων τύπου *Student* του `*this` και του *rhs*.

Ο τελεστής εκχώρησης δηλώνεται ως:

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

και ορίζεται με τη γνωστή συνταγή:

```
EnrolledStudent& EnrolledStudent::operator=( const EnrolledStudent& rhs )
{
   if ( &rhs != this )
   {
      try {  EnrolledStudent tmp( rhs );
             swap( tmp );  }
      catch( EnrolledStudentXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
   }
   return *this;
}; // Student( const Student& rhs )
```

Κατά τα άλλα, έχουμε τις απλές μεθόδους:

```
// getters
   unsigned int getWH() const {  return esWH;  }
   unsigned int getNoOfCourses() const {  return esNoOfCourses;  }
   const Course::CourseKey* getCourses() const {  return esCourses;  }
// setters
   void clearCourses() {  esNoOfCourses = 0;  esWH = 0;  }
```

και τις μεθόδους ενός στοιχείου:

```
   bool find1Course( const string& code ) const
                                          {  return ( findNdx(code) >= 0 );  }
   void add1Course( const Course& oneCourse );
   void delete1Course( const Course& oneCourse );
```

Οι μέθοδοι (**public**) *add1Course()* και *delete1Course()* μένουν (σχεδόν) όπως ήταν στην προηγούμενη μορφή. Η μόνη διαφορά είναι η αντικατάσταση του "`sWH`" από το "`esWH`".

Παρόμοια ισχύουν και για τις μεθόδους (**private**) *findNdx()*, *insert1Course()* και *erase1-Course()*: Τα *sIncr*, *sCourses*, *sNoOfCourses*, *sReserved* γίνονται *esIncr*, *esCourses*, *esNoOfCourses*, *esReserved* αντιστοίχως. Δίνουμε ενδεικτικώς την *insert1Course()* όπου αλλάζει και ο τύπος των εξαιρέσεων που ρίχνονται: *EnrolledStudentXptn* αντί για *StudentXptn*.

```
void EnrolledStudent::insert1Course( const Course::CourseKey& aCode )
{
   if ( esReserved <= esNoOfCourses+1 )
   {
      try {  renew( esCourses, esNoOfCourses, esReserved+esIncr );
             esReserved += esIncr;  }
      catch( MyTmpltLibXptn& )
      {
         throw EnrolledStudentXptn( getIdNum(), "insert1Course",
                                    EnrolledStudentXptn::allocFailed );
      }
   }
   esCourses[esNoOfCourses] = aCode;
   ++esNoOfCourses;
} // Student::insert1Course
```

Οι πολυμορφικές μέθοδοι, εκτός της *swap()*, είναι:

```
void EnrolledStudent::save( ostream& bout ) const
{
   Student::save( bout );

   bout.write( reinterpret_cast<const char*>(&esWH), sizeof(esWH) );
   bout.write( reinterpret_cast<const char*>(&esNoOfCourses),
               sizeof(esNoOfCourses) );
   for ( int k(0); k < esNoOfCourses; ++k )
      bout.write( esCourses[k].s, Course::cCodeSz );

   if ( bout.fail() )
      throw StudentXptn( getIdNum(), "save",
                                        StudentXptn::cannotWrite );
} // EnrolledStudent::save
```

```
void EnrolledStudent::load( istream& bin )
{
   EnrolledStudent tmp;

   tmp.Student::load( bin );
   if ( !bin.eof() )
   {
      bin.read( reinterpret_cast<char*>(&tmp.esNoOfCourses),
                sizeof(esNoOfCourses) );
      if ( tmp.esNoOfCourses >= tmp.esReserved )
      {
         delete[] tmp.esCourses;
         try
         {
            tmp.esCourses = new Course::CourseKey[
                            ((tmp.esNoOfCourses/esIncr)+1)*esIncr ];
            tmp.esReserved = ((tmp.esNoOfCourses/esIncr)+1)*esIncr;
         }
         catch( bad_alloc )
         {
            throw EnrolledStudentXptn( tmp.getIdNum(), "load",
                                       EnrolledStudentXptn::allocFailed );
         }
      }
      for ( int k(0); k < tmp.esNoOfCourses; ++k )
         bin.read( tmp.esCourses[k].s, Course::cCodeSz );
      if ( bin.fail() )
         throw StudentXptn( tmp.getIdNum(), "load", StudentXptn::cannotRead );
      swap( tmp );
   }
} // EnrolledStudent::load

void EnrolledStudent::display( ostream& tout )
{
   Student::display( tout );
   tout << esWH << '\n' << esNoOfCourses << endl;
   for ( int k(0); k < esNoOfCourses; ++k )
      tout << esCourses[k].s << endl;
} // EnrolledStudent::display
```

Και οι τρεις στηρίζονται (με τις "`Student::save(bout)`", "`tmp.Student:: load(bin)`", "`Student::display(tout)`") στις αντίστοιχες μεθόδους της βασικής κλάσης.

Όπως βλέπεις, δεν υπήρξε ανάγκη για πρόσβαση σε κάποιο μέλος της βασικής κλάσης ώστε να το δηλώσουμε "**protected**". Για την ακρίβεια μας χρειάστηκε το *sIdNum* στις ρίψεις εξαιρέσεων αλλά

- η *getIdNum()* είναι "**inline**",
- στη ρίψη εξαίρεσης δεν «βιαζόμαστε»!

Αλλά, δεν τελειώσαμε ακόμη· στις νέες προδιαγραφές του προγράμματος έχουμε: «*Παρομοίως, στο μητρώο φοιτητών ένας φοιτητής που έχει εγγραφεί σε ένα τουλάχιστον μάθημα εμφανίζεται με αντικείμενο EnrolledStudent· αλλιώς εμφανίζεται με αντικείμενο κλάσης Student.*» Ας σκεφτούμε λοιπόν το εξής σενάριο: Θεωρούμε ότι διαβάζουμε –από το `enrllmnt.txt`– την τιμή ενός αντικειμενου της βασικής κλάσης *Student*, που το δείχνει το βέλος `scArr[k]`· διαβάζουμε λοιπόν με τη *readPartFromText()* στο `*scArr[k]`. Μετά διαβάζουμε τον αριθμό μαθημάτων. Αν είναι θετικός θα πρέπει να «αλλάξουμε» τον τύπο του αντικειμένου ώστε να μπορεί να δεχτεί τους κωδικούς μαθημάτων. Όπως στον πίνακα μαθημάτων έτσι και εδώ θα κάνουμε το εξής:

```
   EnrolledStudent* pOneStudent;
// . . .
      try {  pOneStudent = new EnrolledStudent( *scArr[ndx] );  }
      catch( bad_alloc& )
      {   throw . . .     }
```

```
      delete scArr[ndx];
      scArr[ndx] = pOneStudent;
```

Για τη "`new EnrolledStudent(*scArr[k])`" θα χρειαστούμε έναν δημιουργό μετατροπής:

```
EnrolledStudent::EnrolledStudent( const Student& rhs )
   :  Student( rhs ),  esWH( 0 )
{
   try  {  esCourses = new Course::CourseKey[ esIncr ];  }
   catch( bad_alloc )
   {  throw EnrolledStudentXptn( getIdNum(), "EnrolledStudent",
                                 EnrolledStudentXptn::allocFailed );  }
   esReserved = esIncr;
   esNoOfCourses = 0;
} // EnrolledStudent::EnrolledStudent
```

Ο ορισμός αυτού του δημιουργού μαζί με τον τελεστή εκχώρησης της κλάσης «νομιμοποιεί» και εδώ, όπως στην περίπτωση της *OfferedCourse*, την εκχώρηση "αντικείμενο *EnrolledStudent* = αντικείμενο *Student*"· πρίν από την εκχώρηση καλείται αυτός ο δημιουργός για να μετατρέψει το αντικείμενο *Student* σε αντικείμενο *EnrolledStudent* και έτσι εκτελείται η εκχώρηση

"αντικείμενο *EnrolledStudent* = *EnrolledStudent(* αντικείμενο *Student)*"

Και να πώς έγινε η

```
class EnrolledStudent : public Student
{
public:
// constructors, destructor
   explicit EnrolledStudent( int aIdNum=0 );
   EnrolledStudent( const EnrolledStudent& rhs );
   EnrolledStudent( const Student& rhs );
   virtual ~EnrolledStudent() {  delete[] esCourses;  };
// copy assignement
   EnrolledStudent& operator=( const EnrolledStudent& rhs );
// getters
   unsigned int getWH() const {  return esWH;  }
   unsigned int getNoOfCourses() const {  return esNoOfCourses;  }
   const Course::CourseKey* getCourses() const
                                          {  return esCourses;  }
// setters
   void clearCourses() {  esNoOfCourses = 0;  esWH = 0;  }
// 1 Course methods
   bool find1Course( const string& code ) const
                               {  return ( findNdx(code) >= 0 );  }
   void add1Course( const Course& oneCourse );
   void delete1Course( const Course& oneCourse );
// other methods
   virtual void swap( EnrolledStudent& rhs );
   virtual void save( ostream& bout ) const;
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
private:
   enum { esIncr = 3 };
   unsigned int       esWH;
   unsigned int       esNoOfCourses;
   Course::CourseKey* esCourses;
   unsigned int       esReserved;

   int findNdx( const string& code ) const;
   void insert1Course( const Course::CourseKey& aCode );
   void erase1Course( int ndx );
}; // EnrolledStudent

typedef EnrolledStudent* PEnrolledStudent;
```

Και εδώ δεν χρειάζεται να κάνουμε επιφόρτωση των "!=" και "==" αφού μας καλύπτουν οι επιφορτώσεις που κάναμε για τη βασική κλάση.

Η κλάση εξαιρέσεων θα είναι:

```
struct EnrolledStudentXptn :  public StudentXptn
{
   enum { allocFailed=20 };
   unsigned int objKey;
   char         funcName[100];
   int          errorCode;
   char         errStrVal[100];
   int          errIntVal;
   EnrolledStudentXptn( int obk, const char* mn, int ec, const char* sv="" )
      :  StudentXptn( obk, mn, 0, sv ) {  errorCode = ec;  }
   EnrolledStudentXptn( int obk, const char* mn, int ec, int ev )
      :  StudentXptn( obk, mn, 0, ev ) {  errorCode = ec;  }
}; // EnrolledStudentXptn
```

Prj06.6 Η Κλάση *StudentCollection*

Και στην κλάση *StudentCollection* η βασική αλλαγή είναι στη δήλωση του δυναμικού πίνακα που από

```
   Student*   scArr;
```

γίνεται πίνακας βελών:

```
   PStudent*  scArr;
```

Και εδώ θα ξεκινήσουμε από τη *findNdx()* και θα δώσουμε την ίδια λύση που δώσαμε στην *CourseCollection*:

```
int StudentCollection::findNdx( int aIdNum ) const
{
   int ndx;
   if ( aIdNum <= 0 )
      ndx = -1;
   else
      ndx = linSearchP( scArr, scNOfStudents, 0, scNOfStudents-1,
                        Student(aIdNum) );
   return ndx;
} // StudentCollection::findNdx
```

Συνεχίζουμε με τον ερήμην δημιουργό και τον καταστροφέα που –και αυτοί– μοιάζουν με τους αντίστοιχους της *CourseCollection*:

```
StudentCollection::StudentCollection()
{
   try
   {
      scReserved = scIncr;
      scArr = new PStudent[ scReserved ];
      scNOfStudents = 0;
      scPAllEnrollments = 0;
      scArr[scNOfStudents] = new Student;
      for ( int k(scNOfStudents+1); k < scReserved; ++k )
         scArr[k] = 0;
   }
   catch( bad_alloc& )
   {
      throw StudentCollectionXptn( "StudentCollection",
                                   StudentCollectionXptn::allocFailed);
   }
} // StudentCollection::StudentCollection

StudentCollection::~StudentCollection()
{
   for ( int k(0); k <= scNOfStudents; ++k )
```

```
      delete scArr[k];
   delete[] scArr;
} // StudentCollection::~StudentCollection
```

«Αντιγράφοντας» τις *erase1Course()* και *insert1Course()* της *CourseCollection*, παίρνουμε τις δύο μεθόδους «χαμηλού επιπέδου», *erase1Student()* και *insert1Student()* αντιστοίχως:

```
void StudentCollection::erase1Student( int ndx )
{
   delete scArr[ndx];
   scArr[ndx] = scArr[scNOfStudents-1];
   scArr[scNOfStudents-1] = scArr[scNOfStudents];
   scArr[scNOfStudents] = 0;
   --scNOfStudents;
} // StudentCollection::erase1Student

void StudentCollection::insert1Student( Student* pStudent )
{
   if ( scReserved <= scNOfStudents+1 )
   {
      try
      {
         renew( scArr, scNOfStudents+1, scReserved+scIncr );
         scArr[scNOfStudents+1] = scArr[scNOfStudents];      // αλλαγή φρουρού
         for ( int k(scNOfStudents+2); k < scReserved+scIncr; ++k )
            scArr[k] = 0;
         scReserved += scIncr;
      }
      catch( MyTmpltLibXptn& )
      {  throw StudentCollectionXptn( "insert1Student",
                                      StudentCollectionXptn::allocFailed );  }
   }
   scArr[scNOfStudents+1] = scArr[scNOfStudents];
   scArr[scNOfStudents] = pStudent;
   ++scNOfStudents;
} // StudentCollection::insert1Student
```

Πρόσεξε ότι και εδώ –όπως στην *CourseCollection::insert1Course*– αλλάξαμε τον τύπο της παραμέτρου σε "`Student* pStudent`" και δεν μας απασχολεί ο τύπος του αντικειμένου που δείχνει το βέλος: με αυτό θα ασχοληθεί η *add1Student()* που καλεί την *insert1Student()*.

```
void StudentCollection::add1Student( const Student& aStudent )
{
   int ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx < 0 ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      try
      {
         try
         {
            const EnrolledStudent&
                 oneStudent( dynamic_cast<const EnrolledStudent&>(aStudent) );
            insert1Student( new EnrolledStudent(oneStudent) );
            const Course::CourseKey* aStCourses( aStudent.getCourses() );
            for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
            {
               scPAllEnrollments->add1StudentInCourse(
                                         StudentInCourse(aStudent.getIdNum(),
                                                         aStCourses[k].s) );
            }
         }
         catch( bad_cast )
         {  insert1Student( new Student(aStudent) );  }
      }
      catch( bad_alloc )
```

```
      { throw StudentCollectionXptn( "add1Student",
                                     StudentCollectionXptn::allocFailed ); }
   }
} // StudentCollection::add1Student
```

Όσο για τη *StudentCollection*::*delete1Student()*, θα την αλλάξουμε όπως αλλάξαμε και την *CourseCollection*::*delete1Course()*: Θα διαγράφουμε έναν φοιτητή αν έχουμε για αυτόν

- αντικείμενο κλάσης *Student* ή
- αντικείμενο κλάσης *EnrolledStudent* αλλά με *getNoOfCourses()* == 0.

```
void StudentCollection::delete1Student( int aIdNum )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx >= 0 )  // υπάρχει
   {
      EnrolledStudent* pOneStudent(
                       dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
      if ( pOneStudent != 0 )  // EnrolledStudent
      {
         if ( pOneStudent.getNoOfCourses() > 0 )
            throw StudentCollectionXptn( "delete1Student",
                                         StudentCollectionXptn::enrollRef,
                                               aIdNum );
      }
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student
```

Πρόσεξε ότι –όταν το αντικείμενο είναι κλάσης *EnrolledStudent*– δεν μπορούμε να εξετάσουμε τον αριθμό των μαθημάτων από το `scArr[ndx]`· πρέπει να χρησιμοποιήσουμε το `pOneStudent`.

Τώρα θα δούμε πώς διορθώνουμε τις *StudentCollection*::*add1Course()* και *StudentCollection*::*delete1Course()* όπως διορθώσαμε τις *CourseCollection*::*add1Student()* και *CourseCollection*::*delete1Student()*.

Στην §Prj06.5 –στην ανάδειξη της ανάγκης για δημιουργό αντικειμένου *EnrolledStudent* από αντικείμενο κλάσης *Student*– είχαμε δει ένα σενάριο χρήσης αυτού του δημιουργού· θα το χρησιμοποιήσουμε στην *add1Course()*:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   EnrolledStudent* pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
   if ( pOneStudent == 0 )
   {
      try {  pOneStudent = new EnrolledStudent( *scArr[ndx] );  }
      catch( bad_alloc& )
      {
         throw StudentCollectionXptn( "add1Course",
                                      StudentCollectionXptn::allocFailed );
      }
      delete scArr[ndx];
      scArr[ndx] = pOneStudent;
   }
   pOneStudent->add1Course( aCourse );
} // StudentCollection::add1Course
```

Όπως βλέπεις, μετατρέψαμε καταλλήλως την *CourseCollection*::*add1Student()*· παραπέμπουμε στον σχολιασμό της για οτιδήποτε δεν καταλαβαίνεις. Θα επισημάνουμε όμως ότι η μέθοδος αυτή αν κληθεί να καταχωρίσει μάθημα σε αντικείμενο τύπου *Student* θα κάνει τη μετατροπή του σε αντικείμενο *EnrolledStudent* χωρίς να κάνει οτιδήποτε το πρόγραμμα που την καλεί.

```
void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   int ndx( findNdx(aIdNum) );
   if ( ndx < 0 )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   EnrolledStudent* pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[ndx]) );
   if ( pOneStudent != 0 )
   {
      pOneStudent->delete1Course( aCourse );
   // τα παρακάτω μπορεί και να περισσεύουν . . .
      if ( pOneStudent->getNoOfCourses() == 0 )
      {
         Student* tmp( new Student(*pOneStudent) );
         delete pOneStudent;
         scArr[ndx] = tmp;
      }
   }
} // StudentCollection::delete1Course
```

Και αυτή προέρχεται από μεταροπή της *CourseCollection::delete1Student()* και σε παραπέμπουμε εκεί για να ξαναθυμηθείς γιατί «*τα παρακάτω μπορεί και να περισσεύουν . . .*»

Από μετατροπές των μεθόδων *CourseCollection::save()* και *CourseCollection::load()* θα πάρουμε τις *StudentCollection::save()* και *StudentCollection::load()* αντιστοίχως, αφού προηγουμενως ορίσουμε:

```
struct SIndexEntry
{
   unsigned int sIdNum;
   size_t       loc;
   short int    inhLvl;
   explicit SIndexEntry( int aIdNum=0, int aLoc=0, short int aInhL=0 )
      : sIdNum( aIdNum ),  loc( aLoc ),  inhLvl( aInhL )  {  }
}; // SIndexEntry

typedef SIndexEntry* PSIndexEntry;

bool operator!=( const SIndexEntry& lhs, const SIndexEntry& rhs )
{  return ( lhs.sIdNum != rhs.sIdNum );  }

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

Η *SIndexEntry* διαφέρει από τη *CIndexEntry* μόνο στον τύπο του κλειδιού.

```
void StudentCollection::save( ofstream& bout, SIndexEntry* stdntIndex ) const
{
   if ( bout.fail() )
      throw  StudentCollectionXptn( "save",
                                    StudentCollectionXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      stdntIndex[k].sIdNum = scArr[k]->getIdNum();
      stdntIndex[k].loc = bout.tellp();
      EnrolledStudent*
                      pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[k]) );
      if ( pOneStudent == 0 )  // Student
      {  stdntIndex[k].inhLvl = 0;
         scArr[k]->save( bout );  }
      else                     // EnrolledStudent
      {  stdntIndex[k].inhLvl = 1;
         pOneStudent->save( bout );  }
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
```

```
} // StudentCollection::save

void StudentCollection::load( ifstream& bin, const SIndexEntry* stdntIndex )
{
   unsigned int n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(n) );
   if ( !bin.eof() )
   {
      StudentCollection tmp;
      try
      {
         for ( int k(0); k < n && !bin.fail(); ++k )
         {
            Student* pOneStudent;
            if ( stdntIndex[k].inhLvl == 0 ) // Student
               pOneStudent = new Student;
            else                             // EnrolledStudent
               pOneStudent = new EnrolledStudent;
            pOneStudent->load( bin );
            tmp.insert1Student( pOneStudent );
         }
      }
      catch( bad_alloc& )
      {
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::allocFailed );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Τέλος, στην *checkWH()* θα πρέπει να κάνουμε μια μικρή αλλαγή: Θα ελέγχουμε τον εβδομαδιαίο φόρτο εργασίας μόνο για τα `*scArr[k]` που είναι κλάσης *EnrolledStudent*.

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( int k(0); k < scNOfStudents; ++k )
   {
      EnrolledStudent*
                     pOneStudent( dynamic_cast<EnrolledStudent*>(scArr[k]) );
      if ( pOneStudent != 0 )
      {
         if ( pOneStudent->getWH() > maxWH )
            log << "student with id num " << pOneStudent->getIdNum()
                << ": " << pOneStudent->getWH() << " hours/week"
                << endl;
      }
   } // for
} // StudentCollection::checkWH
```

Prj06.7 *StudentInCourse* και *StudentInCourseCollection*

Στην κλάση *StudentInCourse* δεν αλλάζει κάτι· αλλά τώρα είναι μια κλάση συσχέτισης των κλάσεων *OfferedCourse* και *EnrolledStudent*. Δες το Σχ. Prj06-1.

Ούτε η *StudentInCourseCollection* έχει αλλαγές!

Prj06.8 Το 1ο Πρόγραμμα (Δημιουργία Αρχείου)

Ξεκινούμε με αυτό που δεν αλλάζει από το πρόγραμμα του Project 4: η *readStudentData()* παραμένει όπως ήταν. Φυσικά, η δομή των *allStudents* και *allCourses* είναι διαφορετική.

Η *loadCourses()* παραμένει περίπου όπως ήταν: διαγράφεται μόνον η εντολή "`oneCourse.clearStudents();`" αφού τώρα για τα αντικείμενα κλάσης *Course* δεν υπάρχει μέθοδος *clearStudents()*.

Η διαφορά βρίσκεται στη φύλαξη των συλλογών. Κατ' αρχάς έχουμε δύο ευρετήρια· έτσι στη `main()` έχουμε:

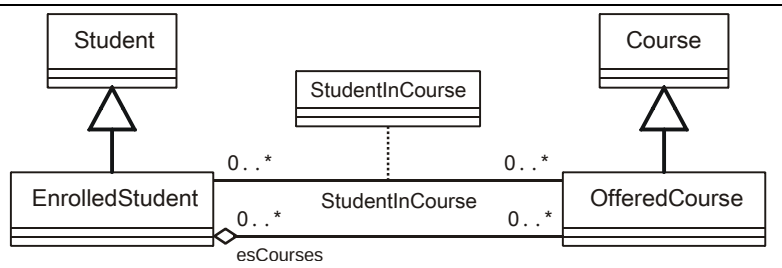
```
      CIndexEntry* cIndex;
      SIndexEntry* sIndex;
      try
      {  cIndex = new CIndexEntry[allCourses.getNOfCourses()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      try
      {  sIndex = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  delete[] cIndex;
         throw ProgXptn( "main", ProgXptn::allocFailed );  }
```

και η κλήση της *saveCollections()* γίνεται με τη

```
      saveCollections( allCourses, cIndex, allStudents, sIndex,
                       allEnrollments );
```

όπου:

```
void saveCollections( CourseCollection& allCourses,
                      CIndexEntry* cIndex,
                      StudentCollection& allStudents,
                      SIndexEntry* sIndex,
                      StudentInCourseCollection& allEnrollments )
{
   ofstream bout( "Courses.dta", ios_base::binary );
   allCourses.save( bout, cIndex );
   bout.close();
   bout.open( "courses.ndx", ios_base::binary );
//   bout.write( reinterpret_cast<const char*>(cIndex),
//               allCourses.getNOfCourses()*sizeof(CIndexEntry) );
   for ( int k(0); k < allCourses.getNOfCourses(); ++k )
      bout.write( reinterpret_cast<const char*>(&cIndex[k]),
                  sizeof(CIndexEntry) );
   bout.close();
   bout.open( "students.dta", ios_base::binary );
   allStudents.save( bout, sIndex );
   bout.close();
   bout.open( "students.ndx", ios_base::binary );
//   bout.write( reinterpret_cast<const char*>(sIndex),
//               allStudents.getNOfStudents()*sizeof(SIndexEntry) );
   for ( int k(0); k < allStudents.getNOfStudents(); ++k )
      bout.write( reinterpret_cast<const char*>(&sIndex[k]),
```



Σχ. Prj06-1 Η συσχέτιση *StudentInCourse* συσχετίζει τις κλάσεις *EnrolledStudent* (φοιτητές που έχουν εγγραφεί σε μαθήματα) και *OfferedCourse* (μαθήματα που έχουν δηλωθεί από φοιτητές). Αυτές κληρονομούν τις *Student* και *Course* αντιστοίχως.

```
                    sizeof(SIndexEntry) );
   bout.close();
   bout.open( "enrllmnt.dta", ios_base::binary );
   allEnrollments.save( bout );
   bout.close();
} // saveCollections
```

Θα έχουμε λοιπόν:

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

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "OfferedCourse.cpp"
#include "Student.cpp"
#include "EnrolledStudent.cpp"
#include "StudentInCourse.cpp"
#include "StudentInCourseCollection.h"
#include "CIndexEntry.h"
#include "CourseCollection.h"
#include "SIndexEntry.h"
#include "StudentCollection.h"
#include "CourseCollection.cpp"
#include "StudentCollection.cpp"
#include "StudentInCourseCollection.cpp"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadCourses( string flNm, CourseCollection& allCourses );
void readStudentData( string flNm,
                      StudentCollection& allStudents,
                      StudentInCourseCollection& allEnrollments,
                      CourseCollection& allCourses,
                      ofstream& log );
void saveCollections( CourseCollection& allCourses,
                      CIndexEntry* cIndex,
                      StudentCollection& allStudents,
                      SIndexEntry* sIndex,
                      StudentInCourseCollection& allEnrollments );

int main()
{
   OfferedCourse oc;
   try
   {
      CourseCollection allCourses;
      StudentCollection allStudents;
      StudentInCourseCollection allEnrollments;

      allEnrollments.setPAllStudents( &allStudents );
      allEnrollments.setPAllCourses( &allCourses );
```

```
      allCourses.setPAllEnrollments( &allEnrollments );
      allStudents.setPAllEnrollments( &allEnrollments );

      loadCourses( "gCourses.dta", allCourses );

      ofstream log( "log.txt" );
      if ( log.fail() )
         throw ProgXptn( "main", ProgXptn::cannotOpen, "log.txt" );

      readStudentData( "enrllmnt.txt", allStudents, allEnrollments,
                       allCourses, log );

      unsigned int maxWH( 30 ); // μέγιστος επιτρεπόμενος φόρτος
                                // φοιτητή: 30 ώρες/εβδομάδα
      allStudents.checkWH( log, maxWH );

      log.close();

      CIndexEntry* cIndex;
      SIndexEntry* sIndex;
      try
      {  cIndex = new CIndexEntry[allCourses.getNOfCourses()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      try
      {  sIndex = new SIndexEntry[allStudents.getNOfStudents()];  }
      catch( bad_alloc )
      {  delete[] cIndex;
         throw ProgXptn( "main", ProgXptn::allocFailed );  }

      saveCollections( allCourses, cIndex, allStudents, sIndex,
                       allEnrollments );

      delete[] sIndex;
      delete[] cIndex;
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( OfferedCourseXptn& x )  { /* . . . */ }
   catch( CourseXptn& x )  { /* . . . */ }
   catch( EnrolledStudentXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( StudentInCourseXptn& x )  { /* . . . */ }
   catch( CourseCollectionXptn& x )  { /* . . . */ }
   catch( StudentCollectionXptn& x )  { /* . . . */ }
   catch( StudentInCourseCollectionXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Prj06.9 Το 2ο Πρόγραμμα (Χρήση Αρχείου)

Στο πρόγραμμα χρήσης του αρχείου, στη `main()`, αλλάζουμε τη δήλωση

```
   IndexEntry* index;
```

σε

```
   SIndexEntry* sIndex;
```

Παρόμοια αλλαγή θα γίνει και στις δύο συναρτήσεις:

```
void loadIndex( string flNm,
                PSIndexEntry& sIndex, unsigned int& ndxSz );
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz );
```

Και στη μεν *loadIndex()* δεν κάνουμε άλλες αλλαγές· δεν ισχύει όμως το ίδιο και για τη *retrieve()*:

```
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz )
{
   ifstream bin( flNm.c_str(), ios_base::binary );
   if ( bin.fail() )
      throw ProgXptn( "retrieve", ProgXptn::cannotOpen,
                      flNm.c_str() );
   string line;
   cout << "Student Id Number: ";  getline( cin, line, '\n' );
   while ( line != "ΤΕΛΟΣ" )
   {
      int idNum( atoi(line.c_str()) );
      int ndx( linSearch(sIndex, ndxSz, 0, ndxSz-1,
                         SIndexEntry(idNum)) );
      if ( ndx < 0 )
         cout << "unknown Student Id Number" << endl;
      else // ndx >= 0
      {
         Student* pOneStudent;
         try {
            if ( sIndex[ndx].inhLvl == 0 )
               pOneStudent = new Student;
            else // == 1
               pOneStudent = new EnrolledStudent;
         }
         catch( bad_alloc& )
         {
            bin.close();
            throw ProgXptn( "retrieve", ProgXptn::allocFailed );
         }
         bin.seekg( sIndex[ndx].loc );
         pOneStudent->load( bin );
         pOneStudent->display( cout );
         delete pOneStudent;  pOneStudent = 0;
      }
      cout << "Student Id Number: ";  getline( cin, line, '\n' );
   } // while
   bin.close();
} // retrieve
```

Πρόσεξε τώρα πώς χρησιμοποιούμε την πληροφορία για τον τύπο του αντικειμένου που έχουμε στο μέλος *inhLvl*:

- Αν –μετά την κλήση της *linSearch()* για αναζήτηση του *idNum* στον sIndex– η *ndx* πάρει μη αρνητική τιμή ξέρουμε ότι υπάρχει το αντικείμενο που ψάχνουμε. Το `sIndex[ndx].loc` μας λέει πού βρίσκεται.
- Το νέο ερώτημα που προκύπτει είναι: ποιος ο τύπος του αντικειμένου ή ποια *load()* να χρησιμοποιήσω; Αν το `sIndex[ndx].inhLvl` έχει τιμή "`0`" το αντικείμενο είναι τύπου *Student*· αλλιώς, αν έχει τιμή "`1`", είναι τύπου *EnrolledStudent*.
- Και πώς χρησιμοποιούμε αυτό που μάθαμε; Όπως βλέπεις, έχουμε δηλώσει ένα βέλος "`Student* pOneStudent`" που μπορεί να δείχνει αντικείμενα κλάσης *Student* και των παραγώγων της. Στην περίπτωσή μας έχουμε μια μόνον παράγωγη, την *EnrolledStudent*:

```
            if ( sIndex[ndx].inhLvl == 0 )
               pOneStudent = new Student;
            else // == 1
               pOneStudent = new EnrolledStudent;
```

- Μετά από αυτό οι εντολές:

```
         pOneStudent->load( bin );
         pOneStudent->display( cout );
```

θα χρησιμοποιήσουν τις *σωστές* μεθόδους αφού στη *Student* (αλλά και στην *EnrolledStudent*) έχουμε δηλώσει:

```
   virtual void load( istream& bin );
   virtual void display( ostream& tout );
```

Ακόμη, στη "`delete pOneStudent`", θα κληθεί ο σωστός καταστροφέας αφού έχουμε δηλώσει

```
virtual ~Student() { };
```

(και "`virtual ~EnrolledStudent() { delete[] esCourses; }`".)

Η `main()` είναι:

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

#include "MyTmpltLib.h"

using namespace std;

#include "Course.cpp"
#include "Student.cpp"
#include "EnrolledStudent.cpp"
#include "SIndexEntry.h"

struct ProgXptn
{
   enum { allocFailed, cannotOpen };
   char functionName[100];
   int  errorCode;
   char errStrVal[100];
   ProgXptn( const char* fn, int ec, const char* sv="" )
      :  errorCode( ec )
   {  strncpy( functionName, fn, 99 );  functionName[99] = '\0';
      strncpy( errStrVal, sv, 99 );  errStrVal[99] = '\0';  }
}; // ProgXptn

void loadIndex( string flNm,
                PSIndexEntry& sIndex, unsigned int& ndxSz );
void retrieve( string flNm, PSIndexEntry sIndex, int ndxSz );

int main()
{
   SIndexEntry* sIndex;
   unsigned int ndxSz;
   try
   {
      loadIndex( "students.ndx", sIndex, ndxSz );
      retrieve( "students.dta", sIndex, ndxSz );
   } // try
   catch( ProgXptn& x )  { /* . . . */ }
   catch( MyTmpltLibXptn& x )  { /* . . . */ }
   catch( EnrolledStudentXptn& x )  { /* . . . */ }
   catch( StudentXptn& x )  { /* . . . */ }
   catch( ... )
   {  cout << "unexpected exceptioon" << endl;  }
} // main
```

Prj06.10Τι Είδαμε σε Αυτό το Παράδειγμα

Κατ' αρχάς θα επαναλάβουμε ότι εδώ *δεν έχουμε* καλή σχεδίαση κλάσεων και ότι «*οι EnrolledStudent και OfferedCourse θα έλυναν τα πρόβλημα αν δεν κληρονομούσαν τις Student και Course (αλλά περιείχαν τα κλειδιά –sIdNum και cCode– για αντιστοίχιση.) Ξαναδιάβασε την §23.14.*»

Τώρα, ας επισημάνουμε αυτά που είδαμε σε αυτό παράδειγμα:

- Είδαμε δύο παραδείγματα κληρονομιάς: *Course* ← *OfferedCourse* (§Prj06.3) και *Student* ← *EnrolledStudent* (§Prj06.5).

- Είδαμε πώς ορίζουμε τους δημιουργούς των παράγωγων κλάσεων από αυτούς των βασικών. Στην περίπτωση της *OfferedCourse* βλέπεις ότι τα πάντα ρυθμίζονται στη λίστα εκκίνησης και τα σώματα των συναρτήσεων είναι κενά (§Prj06.3).
- Είδαμε πώς μπορούμε να ορίσουμε στις παράγωγες κλάσεις τους κατάλληλους *δημιουργούς μετατροπής*

```
OfferedCourse( const Course& oneCourse ); // (§Prj06.3)
EnrolledStudent( const Student& rhs ); // (§Prj06.5)
```

  ώστε, μεταξύ άλλων, να «νομιμοποιήσουμε» και τις εκχωρήσεις

  αντικείμενο *OfferedCourse* = αντικείμενο *Course*
  αντικείμενο *EnrolledStudent* = αντικείμενο *Student*

  χωρίς να επιφορτώσουμε τον σχετικό τελεστή εκχώρησης.
- Είδαμε πώς μπορούμε να βάλουμε στον ίδιο πίνακα αντικείμενα της βασικής και της παράγωγης κλάσης. Πώς; Κάνοντας πίνακα με βέλη προς αντικείμενα της βασικής:

```
PCourse*  ccArr; // (§Prj06.4)
PStudent* scArr; // (§Prj06.6)
```

- Φυσική συνέπεια της παραπάνω λύσης είναι η ανάγκη να μαθαίνουμε τον τύπο του αντικειμένου που δείχνει κάποιο βέλος `ccArr[k]` ή `scArr[k]`. Είδαμε τη λύση αυτού του προβλήματος με *δυναμική τυποθεώρηση*.[3]
- Είδαμε ακόμη και τη χρήση της *δυναμικής τυποθεώρησης* –μαζί με «ανορθόδοξη» διαχείριση εξαίρεσης– για να μάθουμε τον τύπο της πραγματικής παραμετρου που αντιστοιχεί σε τυπική *παράμετρο αναφοράς*.

[3] Δεν είναι κακή ιδέα να δοκιμάσεις και άλλη λύση.

κεφάλαιο

24

«Πέφτεις σε Λάθη...» – Εξαιρέσεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Θα ξαναδούμε –συμπληρωμένα– τα εργαλεία που σου προσφέρει η C++ (και η C) για να διαχειριστείς προβλήματα που παρουσιάζονται όταν εκτελείται το πρόγραμμά σου. Θα δεις τις εξαιρέσεις από την καλή και την ανάποδη:

- Θα κατάλάβεις την υπεροχή του μηχανισμού εξαιρέσεων της C++.
- Θα καταλάβεις όμως ότι πρέπει να ασφαλίζεις το πρόγραμμά σου και από τις εξαιρέσεις που το ίδιο ρίχνει.

Προσδοκώμενα αποτελέσματα:

Εξαιρέσεις χρησιμοποιούμε σε μεγάλη έκταση στα μισά από τα προηγούμενα κεφάλαια. Τι καινούριο θα μάθεις εδώ από πρακτική άποψη;

- Κάποια εργαλεία που θα κάνουν τη διαχείριση (των δικών μας) εξαιρέσεων πιο λειτουργική (τουλάχιστον στον καλύτερο εντοπισμό των προβλημλατων).
- Κανόνες για να σχεδιάσεις δικές σου κλάσεις εξαιρέσεων (αν δεν σου αρέσουν αυτές της C++ και οι δικές μας).
- Περισσότερα πράγματα για την ασφάλεια προς τις εξαιρέσεις.

Έννοιες κλειδιά:

- *συνάρτηση assert()*
- *τερματισμός εκτέλεσης*
- *συνάρτηση exit()*
- *προδιαγραφές εξαιρέσεων*
- *συναρτησιακή ομάδα* `try`
- *κλάσεις εξαιρέσεων*
- *ασφάλεια προς τις εξαιρέσεις*

Περιεχόμενα:

24.1 Τι Έχουμε από τη C 919
- 24.1.1 Η Συνάρτηση *assert()* 919
- 24.1.2 Συναρτήσεις Τερματισμού Εκτέλεσης 920
  - 24.1.2.1 Σχέση `std::exit()` και `return` 921
- 24.1.3 Τι Λάθος Έκανα; *errno* 922

24.2 Μήνυμα με Τιμές Συνάρτησης και Παραμέτρων 924

24.3 Συμπληρώματα στην «Ιστορία με Εξαιρέσεις» 927

24.4 Οι Συναρτήσεις Διαχείρισης Εξαιρέσεων 928
- 24.4.1 Η Συνάρτηση `std::set_terminate()` 928
- 24.4.2 Η Συνάρτηση `std::terminate()` 930
- 24.4.3 * Προδιαγραφές Εξαιρέσεων και Σχετικές Συναρτήσεις 933
- 24.4.4 * Η Συνάρτηση `std::uncaught_exception()` 934

24.5 Συναρτησιακή Ομάδα try 935

24.6 Οι Τύποι Εξαιρέσεων της C++ 936
- 24.6.1 Η Κλάση *logic_error* και οι Παράγωγές της 938

24.6.1.1 Η Κλάση *domain_error* ... 938
24.6.1.2 Η Κλάση *invalid_argument* ... 939
24.6.1.3 Η Κλάση *length_error* ... 940
24.6.1.4 Η Κλάση *out_of_range* ... 940
24.6.2 Η Κλάση *runtime_error* και οι Παράγωγές της ... 941
24.6.2.1 Η Κλάση *range_error* ... 941
24.6.2.2 Οι Κλάσεις *overflow_error* και *underflow_error* ... 941
24.6.3 Να Χρησιμοποιούμε Αυτές τις Κλάσεις; ... 943
24.7 Πώς να Σχεδιάζεις Δικές σου Κλάσεις Εξαιρέσεων ... 943
24.8 Οι Δικές μας Κλάσεις Εξαιρέσεων ... 944
24.8.1 Δύο Μέθοδοι για τις Κλάσεις Εξαιρέσεων ... 945
24.9 Ασφάλεια ως προς τις Εξαιρέσεις ... 948
24.10 Σύνοψη ... 949

Εισαγωγικές Παρατηρήσεις:

Σε πολλά από τα προγράμματα-παραδείγματα που γράψαμε μέχρι τώρα δίναμε ιδιαίτερη προσοχή στη διασφάλιση κάποιων συνθηκών σε ορισμένα σημεία του προγράμματος. Χαρακτηριστικά παραδείγματα:

- εγκυρότητα των στοιχείων εισόδου που είναι μέρος της εξασφάλισης της προϋπόθεσης του προγράμματος,
- εξασφάλιση της αναλλοίωτης πριν από μια επαναληπτική εντολή,
- εξασφάλιση ότι τα ορίσματα μιας συνάρτησης ανήκουν στο πεδίο ορισμού της (εξασφάλιση προϋπόθεσης),
- εξασφάλιση της αναλλοίωτης μιας κλάσης.

Στην αρχή, μάθαμε να αντιδρούμε «βιαίως» αν βρίσκαμε να μην ισχύει κάποια επιθυμητή συνθήκη. Στην §4.3, στο πρόγραμμα της ελεύθερης πτώσης, είχαμε:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
```

Αν η συνθήκη-όρισμα της *assert()* πάρει τιμή `false`, η συνάρτηση βγάζει ένα μήνυμα και διακόπτει την εκτέλεση του προγράμματος.

Στο παράδ. 1 της §5.5 γράφαμε κάπως αλλιώς το ίδιο πρόγραμμα:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   // κάνουμε τους υπολογισμούς
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
```

Αν η τιμή που δίνουμε στην h δεν είναι έγκυρη (αν δηλαδή $h < 0$) τότε βγάζουμε ένα μήνυμα και δεν κάνουμε οτιδήποτε άλλο.

Παρομοίως, στην §7.7 γράφαμε τη

```
   // factorial -- Υπολογίζει το a!
   unsigned long int factorial( int a )
   {
      if ( a < 0 )
      {
         cout << " η factorial κλήθηκε με όρισμα " << a << endl;
         exit( EXIT_FAILURE );
      }
      unsigned long int fv( 1 );
      for ( int k(1);  k <= a;  ++k )  fv *= k;
      return fv;
   } // factorial
```

(ή κάπως έτσι.)

Εδώ, στην περίπτωση που η *factorial()* κληθεί με όρισμα εκτός πεδίου ορισμού (αρνητικό), η συνάρτηση διακόπτει την εκτέλεση του προγράμματος.

Αργότερα μάθαμε να είμαστε πιο ελαστικοί: να δίνουμε τη δυνατότητα στον χρήστη να ξαναδώσει τα (πιθανόν λαθεμένα) στοιχεία εισόδου ή *να ρίχνουμε μια εξαίρεση* αντί να καλέσουμε την *exit()*.

Από τα μέχρι τώρα παραδείγματα θα πρέπει να έχεις αντιληφθεί ότι η αντιμετώπιση των σφαλμάτων στο πρόγραμμα έχει δύο σκέλη:

- Να βρούμε το σφάλμα *εγκαίρως*, πριν μας «κάνει τη ζημιά». Για παράδειγμα, στο πρόγραμμα της ελεύθερης πτώσης (§2.7 και §3.8) αφού η προϋπόθεση λέει ότι θα πρέπει να έχουμε $h \geq 0$ δεν χρειάζεται να φθάσουμε στον υπολογισμό της $\sqrt{2h/g}$.
- Να αντιμετωπίσουμε την κατάσταση. Και πώς γίνεται αυτό;
  - Ένας τρόπος είναι να σταματήσουμε την εκτέλεση του προγράμματος αφού προηγουμένως φυλάξουμε όσα μπορούμε από αυτά που ήδη έκανε. Αυτή η λύση είναι η λιγότερο επιθυμητή.
  - Ένας καλύτερος τρόπος είναι να «καθαρίσουμε» ό,τι έχει «μολυνθεί» από τα λανθασμένα στοιχεία, να φέρουμε το πρόγραμμά μας σε μια αποδεκτή κατάσταση και να συνεχίσουμε από εκεί.

Στο κεφάλαιο αυτό θα συζητήσουμε πιο εκτεταμένα τα εργαλεία που μας δίνει η C++ για να αντιμετωπίζουμε τις «δύσκολες» καταστάσεις. Κατά βάση θα δούμε τις εξαιρέσεις, αλλά πριν από αυτό θα δούμε μερικά εργαλεία που μας δίνει η C, κυρίως αυτά που χρησιμοποιήσαμε ήδη αμέσως ή εμμέσως.

Σημείωση:▶

Πριν προχωρήσουμε θα πρέπει να αναφέρουμε ότι και η C έχει σύστημα εξαιρέσεων με το οποίο όμως δεν θα ασχοληθούμε.◀

24.1 Τι Έχουμε από τη C

Όπως είδαμε η C μας δίνει τη συνάρτηση *assert()* για να ελέγχουμε αν ισχύει κάποια συνθήκη επαλήθευσης σε συγκεκριμένο σημείο του προγράμματός μας.[1] Είδαμε ακόμη τη συνάρτηση *exit()* με την κλήση της οποίας τερματίζεται η εκτέλεση του προγράμματός μας.

24.1.1 Η Συνάρτηση *assert()*

Μπορείς να σκέφτεσαι τη (μακρο)συνάρτηση *assert()* ως:

```
void assert( bool test )
```

Όταν κληθεί, αν το όρισμά της υπολογισθεί σε `true` δεν «αντιδρά». Αν υπολογισθεί `false` τότε δίνει μήνυμα και διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση *abort()*.

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cassert>`".

[1] Για το νέο πρότυπο της C προτάθηκε (με την τεχνική αναφορά ISO/IEC TR 24731-1-2007) μια οικογένεια εργαλείων για τη διαχείριση παραβίασης περιορισμών (constraints) κατά τη διάρκεια της εκτέλεσης. Σε σχέση με τα εργαλεία αυτά η *assert()* είναι μάλλον απλοϊκή.

24.1.2 Συναρτήσεις Τερματισμού Εκτέλεσης

Η συνάρτηση *exit()*: Την είδαμε για πρώτη φορά στην §7.7 (Παράδ. 1)· προκαλεί κανονικό τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void exit( int status )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση:

- Κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την *atexit()*, που θα δούμε στην συνέχεια. Οι συναρτήσεις εκτελούνται με σειρά αντίστροφη της σειράς καταχώρισης.
- Στη συνέχεια, οι ενταμιευτές των ρευμάτων που έχουν ανοιχθεί για γράψιμο γράφονται στα αντίστοιχα αρχεία και όλα τα ρεύματα κλείνουν.
- Τέλος, τερματίζεται η εκτέλεση του προγράμματος και ο έλεγχος περνάει στο ΛΣ. Αν η τιμή της *status* είναι "`0`" ή *EXIT_SUCCESS* στο ΛΣ περνάει η πληροφορία *επιτυχούς τερματισμού* (successful termination). Αν η τιμή της *status* είναι "`1`" ή *EXIT_FAILURE* περνάει η πληροφορία *ανεπιτυχούς τερματισμού* (unsuccessful termination).[2]

Η συνάρτηση *abort()*: Είπαμε παραπάνω ότι η *assert()* μπορεί να «*διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση abort()*.» Η *abort()* προκαλεί *μη-κανονικό* τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void abort( )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση τερματίζεται η εκτέλεση του προγράμματος και στο ΛΣ περνάει η πληροφορία *ανεπιτυχούς τερματισμού*. Το τι θα γίνει με τα ανοιχτά ρεύματα εξαρτάται από τη συγκεκριμένη υλοποίηση.

Η συνάρτηση *_Exit()*: Η δήλωσή της (στο `cstdlib`) είναι (περίπου):

```
void _Exit( int status );
```

Προκαλεί κανονικό τερματισμό του προγράμματος και περνάει στο ΛΣ πληροφορία όπως η *exit()*. Σε σχέση με τα ρεύματα ισχύουν αυτά που είπαμε για την *abort()*.

Η συνάρτηση *atexit()*: Παραπάνω είπαμε ότι όταν κληθεί η *exit()* «*κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την atexit()*».

Η δήλωσή της *atexit()* (στο `cstdlib`) είναι (περίπου):

```
int atexit( void (*func)() );
```

Δηλαδή, η *atexit()* περιμένει ένα όρισμα που είναι βέλος προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την "`void f()`").

Αν η καταχώριση γίνει επιτυχώς η συνάρτηση επιστρέφει "`0`" ενώ σε περίπτωση αποτυχίας επιστρέφεται μη-μηδενική τιμή. Κάθε υλοποίηση της C++ θα πρέπει να επιτρέπει τουλάχιστον 32 καταχωρίσεις.

Δες ένα παράδειγμα:

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

void bye()
{  cout << "Bye bye!" << endl;  }

void thatsAll()
{  cout << "That is all!!" << endl;  }
```

[2] Στο `cstdlib` υπάρχουν οι ορισμοί:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

```
int main()
{
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   if ( atexit(&thatsAll) != 0 )
      cout << "thatsAll registration failed" << endl;

   exit(0);
}
```

Αποτέλεσμα:

```
That is all!!
Bye bye!
```

Πρόσεξε τα εξής:

1. Πρώτα έγινε η καταχώριση της *bye()* και μετά της *thatsAll()*· αλλά κατά τον τερματισμό εκτελείται πρώτα η *thatsAll()* και μετά η *bye()*.
2. Όπως μάθαμε στην §14.3, μπορούμε να μην βάλουμε το "`&`" και να γράψουμε πιο απλά:

```
   if ( atexit(bye) != 0 ) . . .
   if ( atexit(thatsAll) != 0 ) . . .
```

Προσοχή! Αν στο πρόγραμμά σου δεν βάλεις "`using namespace std`" θα πρέπει να χρησιμοποιείς το πρόθεμα "`std::`" για όλες τις συναρτήσεις που είδαμε παραπάνω.

24.1.2.1 Σχέση `std::exit()` και `return`

Τι διαφορά υπάρχει μεταξύ τερματισμού με *exit()* και τερματισμού με "`return`" στη `main`;

- Αν δώσεις "`return` *n*" θα καταστραφούν –με κλήση των αντίστοιχων καταστροφέων– όλα τα αντικείμενα της `main` που έχουν δημιουργηθεί στη στοίβα και μετά θα εκτελεσθεί η κλήση "`exit( n )`".
- Αν δώσεις "`exit( n )`" δεν θα κληθούν οι καταστροφείς· έτσι, αν περιμένεις κάτι από αυτούς θα πρέπει να φροντίσεις να γίνει με άλλον τρόπο.

Να ένα παράδειγμα:

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

struct B
{
   B( int k ) :  bM( k ) { };
   ~B()
   {  cout << "destruction of a B object: " << bM << endl;  }
private:
   int bM;
}; // B

B b1( 3 );

int main()
{
   B b2( 7 );

   return 0;
}
```

Αυτό το πρόγραμμα θα σου δώσει:

```
destruction of a B object: 7
destruction of a B object: 3
```

Αν αλλάξεις το "`return 0`" σε "`exit( 0 )`" θα πάρεις:

```
destruction of a B object: 3
```

Δηλαδή καλείται ο καταστροφέας να καταστρέψει το *b1* που υλοποιείται σε *στατική* μνήμη αλλά όχι το *b2* που υλοποιείται σε μνήμη στοίβας.[3]

Ωραία όλα αυτά, αλλά αφού και στις δύο περιπτώσεις θα σταματήσει η εκτέλεση του προγράμματος θα «καταστραφούν» τα πάντα! Τι θα μπορούσε να περιμένει από έναν καταστροφέα ένα *καλό* πρόγραμμα; Μια απάντηση μπορεί να είναι η εξής: Να «καθαρίζει» τα αντικείμενα από ευαίσθητες πληροφορίες (ξαναδές την §16.14) πριν επιστραφεί η μνήμη τους στη στοίβα ή στον σωρό!

24.1.3 Τι Λάθος Έκανα; *errno*

Αν έχουμε κάποιο πρόβλημα όταν καλούμε μια συνάρτηση από τις βιβλιοθήκες της C (π.χ.: με λάθος τιμές παραμέτρων) μπορεί να πάρουμε μήνυμα λάθους μέσω της καθολικής μεταβλητής *errno*. Ας ξαναγράψουμε το πρόγραμμα της ελεύθερης πτώσης:

```
#include <iostream>
#include <cmath>
#include <cerrno>
#include <string>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
// προσπάθησε να υπολογίσεις το tP
   errno = 0;
   tP = sqrt( (2/g)*h );
   if ( errno != EDOM )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX) && (tP ≈ √(2h/g))
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   // Τύπωσε τα tP, vP
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   }
   else // errno == EDOM
      cout << strerror( errno )
           << ": το ύψος δεν μπορεί να είναι αρνητικό" << endl;
}
```

Να δύο παραδείγματα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: 80
 Αρχικό ύψος = 80 m
 Χρόνος Πτώσης = 4.03855 sec
 Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
 Δώσε μου το αρχικό ύψος σε m: -10
Domain error: το ύψος δεν μπορεί να είναι αρνητικό
```

Τι είναι η *errno*; Μπορείς να τη σκέφτεσαι ως μια καθολική μεταβλητή που έχει δηλωθεί «πριν από σένα για σένα» ως[4]

```
int errno;
```

---

[3] Φυσικά, αν δηλώσεις "`static B b2( 7 )`" θα κληθεί ο καταστροφέας και για αυτό.

[4] Το πρότυπο της C λέει ότι μπορεί να είναι κάποιος μακροορισμός που καταλήγει σε μια *τροποποιήσιμη τιμή-l*.

Όταν αρχίζει η εκτέλεση του προγράμματος η τιμή της *errno* είναι "`0`". Ορισμένες συναρτήσεις βάζουν στην *errno* θετική τιμή *αλλά δεν υπάρχει συνάρτηση που να τη μηδενίζει*.

- Ορισμένες συναρτήσεις, μεταξύ αυτών και οι συναρτήσεις της math, αν τροφοδοτηθούν με ορίσματα εκτός πεδίου ορισμού βάζουν στην *errno* τιμή *EDOM* (ορίζεται στο `cerrno`).
- Ορισμένες συναρτήσεις όταν υπολογίσουν εξαιρετικώς μεγάλη τιμή (που δεν παριστάνεται στον τύπο του αποτελέσματος) βάζουν στην *errno* τιμή *ERANGE*.

Τι κάναμε λοιπόν εδώ; Βάλαμε στην *errno* τιμή "`0`" και –χωρίς να ελέγξουμε την τιμή της *h*– δώσαμε "`tP=sqrt((2/g)*h)`". Μετά από αυτό ελέγχουμε την τιμή της *errno* και προχωρούμε μόνο στην περίπτωση που `errno != EDOM`. Ενώ στο αρχικό πρόγραμμα προσέχαμε «*να βρούμε το σφάλμα εγκαίρως, πριν μας "κάνει τη ζημιά"*» εδώ προχωρήσαμε στο επικίνδυνο βήμα και μετά είπαμε «*δεν έπρεπε να το κάνω!*»

Η συνάρτηση *strerror()* παίρνει την τιμή της *errno* και μας δίνει το αντίστοιχο μήνυμα. Στην περίπτωσή μας απεικόνισε το *EDOM* στο "`Domain error`".

Ως δεύτερο παράδειγμα δες το παρακάτω, αφού πρώτα ξαναδείς αυτά που λέγαμε στην §13.8.

```
double d;
char* p;

errno = 0;
d = strtod( "7.1e+318", &p );
if ( (d == HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
errno = 0;
d = strtod( "-7.1e+318", &p );
if ( (d == -HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
```

Αποτέλεσμα:

```
TRUE
TRUE
```

Πράγματι, οι τιμές $\pm 7.1\times 10^{+318}$ δεν μπορούν να παρασταθούν στον τύπο `double` και έτσι παίρνουμε ως αποτέλεσμα `±HUGE_VAL` αντιστοίχως ενώ η *errno* παίρνει τιμή *ERANGE*. Το C99 λέει: «*Αν η σωστή τιμή είναι εκτός της περιοχής τιμών που μπορεί να παρασταθούν, επιστρέφεται συν ή πλην HUGE_VAL και η τιμή του ERANGE αποθηκεύεται στην errno.*» Για τον λόγο αυτόν λέγαμε στην §13.8 ότι πρέπει να ελέγχεις τη συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

Γενικώς, οι συναρτήσεις των βιβλιοθηκών της C σε περίπτωση λάθους

- είτε επιστρέφουν τιμή εκτός πεδίου τιμών (όπως η `HUGE_VAL`),
- είτε επιστρέφουν *ακραία* τιμή εντός του πεδίου τιμών (όπως η `INT_MAX`).

ενώ παραλλήλως μπορεί να δίνουν και κάποια θετική τιμή στην *errno*.

Έτσι, η οδηγία που δίνεται για τη χρήση της *errno* είναι η εξής:

- Πριν από την κλήση της συνάρτησης βάλε την εντολή "`errno = 0`".
- Μετά την κλήση της έλεγξε την τιμή που επίστρεψε η συνάρτηση και την τιμή της *errno*.

Πάντως:

- Το πρότυπο δεν δεσμεύει τις διάφορες υλοποιήσεις να επιστρέφουν σε κάθε περίπτωση αποτυχίας συγκεκριμένες τιμές. Για παράδειγμα: δεν λέει τι τιμή θα επιστρέφει η *sqrt()* σε περίπτωση που θα τροφοδοτηθεί με αρνητικό όρισμα.[5]
- Δεν δεσμεύει τις υλοποιήσεις στο να δίνουν τιμή στην *errno*. Για παράδειγμα, για τη *strtod()* το C99 λέει: «*Αν στο αποτέλεσμα έχουμε υποχείλιση (underflow), η συνάρτηση επιστρέφει τιμή που το μέγεθός της δεν υπερβαίνει τον ελάχιστο κανονικοποιημένο*

[5] Οι περισσότερες υλοποιήσεις θα επιστρέψουν `NaN`.

αριθμό του τύπου επιστροφής· το αν η errno θα πάρει τιμή ERANGE καθορίζεται από την υλοποίηση.»[6]

Όπως καταλαβαίνεις, με αυτόν τον τρόπο δεν είναι εύκολο να γράψεις προγράμματα που θα είναι δυνατόν να μεταφερθούν από τη μια υλοποίηση στην άλλη χωρίς αλλαγές.

Τι θα κάνεις λοιπόν;

- ***Προσπάθησε να αποφεύγεις τη χρήση της errno.***
- ***Αν πρέπει να το χρησιμοποιήσεις υποχρεωτικώς συμβουλέψου το C99 και το εγχειρίδιο χρήσης της C που χρησιμοποιείς.***

24.2 Μήνυμα με Τιμές Συνάρτησης και Παραμέτρων

Τα εργαλεία που μας δίνει η C είναι στην πραγματικότητα αυτά που χρειάζονται οι προγραμματιστές που χρησιμοποιούν γλώσσες χωρίς εργαλεία για τη διαχείριση εξαιρέσεων. Οι συναρτήσεις που υπήρχαν στις παλιές βιβλιοθήκες έκοβαν πολύ εύκολα την εκτέλεση του προγράμματος όταν εβρισκαν κάποιο σοβαρό λάθος. Έτσι και οι προγραμματιστές έγραφαν τα προγράμματά τους με τον ίδιο τρόπο· επομένως, η *exit()* και η *abort()* ήταν εργαλεία πολύ χρήσιμα. Αργότερα τα πράγματα άλλαξαν· ο προγραμματιστής δεν ήθελε να του κόβει το πρόγραμμα η *sqrt()* ή κάποια συνάρτηση που έγραφε ο ίδιος ο προγραμματιστής και έτσι η *exit()* και η *abort()* έπαψαν να είναι τόσο χρήσιμες. Αλλά αυτό είχε ως αντίτιμο το να γεμίσει το πρόγραμμα με **if**· και αυτό μπορεί να μην ήταν ενοχλητικό από την άποψη της ταχύτητας εκτέλεσης –οι ταχύτητες των επεξεργαστών είχαν βελτιωθεί– ήταν όμως ενοχλητικό από την άποψη της πολυπλοκότητας του κώδικα καθώς έτσι είχαμε «ανακάτεμα» του αλγόριθμου με τις άμυνες από πιθανά λάθη.

Οι συναρτήσεις έστελναν μηνύματα για τα διάφορα προβλήματα που έβρισκαν με την τιμή της συνάρτησης, με παραμέτρους **out** ή με καθολικές μεταβλητές σαν την *errno*.

Σε πολλές περιπτώσεις, για να μην αυξάνεται το πλήθος των παραμέτρων, η ίδια παράμετρος χρησιμοποιείται για να μεταφέρει όχι μόνον κωδικό λάθους αλλά και άλλες πληροφορίες. Ήδη, εδώ είδαμε:

- Στη συνάρτηση *bisection()*, στην §14.3, επιστρέφουμε μήνυμα λάθους με μια παράμετρο αναφοράς, την *errCode*. Αυτό το είδος λάθους που έχουμε εκεί πιθανότατα απαιτεί ρίψη εξαίρεσης. Ας πούμε όμως ότι κάνεις την αλλαγή που λέγαμε στις παρατηρήσεις και ζητάς να πετύχεις την $|f(x)| < \varepsilon'$ σε *nMax* επαναλήψεις το πολύ (διότι η $|b - a|/2 < \varepsilon$ μπορεί να μη είναι ικανοποιητική). Στην περίπτωση αυτή βάζουμε και άλλη μια παράμετρο, *nMax*, με την οποία βάζουμε έναν μέγιστο αριθμό επαναλήψεων που επιθυμούμε να γίνουν. Πώς θα γνωστοποιήσει η *bisection()* στη συνάρτηση που την καλεί το εξής γεγονός: "*έκανα nMax επαναλήψεις αλλά η* $|f(x)| < \varepsilon'$ *δεν ισχύει*"; Με εξαίρεση; Όχι βέβαια.
- Στην άσκηση 13-10 σε καλούμε να γράψεις συνάρτηση, την *trinomial()*, που μέσω μιας παρόμοιας παραμέτρου επιστρέφει *το πλήθος των πραγματικών ριζών ή κωδικό λάθους*. Αλλά, ας ξανασκεφτούμε αυτήν την περίπτωση: Το να καλέσουμε τη συνάρτηση για να μας λύσει την εξίσωση "7 = 0" (κωδικός: –1) ή "0 = 0" (κωδικός: *INT_MAX*) σημαίνει πιθανότατα ότι το πρόγραμμά μας έχει κάποιο σοβαρό λάθος. Το να βρούμε ότι το τριώνυμο έχει 0 ή 1 ή 2 πραγματικές ρίζες είναι κάτι το αναμενόμενο. Έτσι, το να εξετάζουμε την τιμή μιας παραμετρου για να καταλάβουμε από την τιμή της αν είναι επιστρεφόμενη τιμή ή κωδικός λάθους μας οδηγεί στο να γράψουμε ένα μπερδεμένο πρόγραμμα. Αν έχεις λύσει την άσκηση, θα έχεις δει ήδη το σχετικό μπέρδεμα και *μέσα* στη συνάρτηση.

[6] Ο μεταγλωττιστής gcc (Dev-C++) βάζει τιμή *ERANGE* στην *errno*.

Θα ήταν λοιπόν καλύτερο να βάλουμε *δύο* παραμέτρους με επιστρεφόμενη τιμή: μια με τον κωδικό σφάλματος και μιαν άλλη για το πλήθος των πραγματικών ριζών. Θα μπορούσαμε ακόμη να μιμηθούμε τη C: να επιστρέφουμε με παράμετρο το πλήθος και να βάζουμε κωδικό λάθους στην *errno* (ή σε κάποια δική μας *myErrNo*.)

Ναι, θα μπορούσαμε να κάνουμε κάτι από αυτά αλλά το μπέρδεμα της διαχείρισης σφαλμάτων με συνήθεις υπολογισμούς παραμένει.

Το πόσο μπερδεμένο μπορεί να είναι ένα πρόγραμμα γραμμένο με αυτόν τον τρόπο φαίνεται όταν έχεις συναρτήσεις που επιστρέφουν μια τιμή. Θα ξαναγράψουμε το πρόγραμμα-παράδειγμα για τους συνδυασμούς που είδαμε στην §14.9.1 με τις εξής απαιτήσεις:

- Κάθε μια από τις συναρτήσεις *comb()*, *factorial()* και *natProduct()* θα πρέπει να αποφεύγει υπολογισμούς με ορίσματα εκτός πεδίου ορισμού, ενώ σε τέτοια περίπτωση θα πρέπει να επιστρέφει κάποιο μήνυμα λάθους. Επειδή οι συναρτήσεις αυτές είναι χρήσιμες γενικώς, θα πρέπει να έχουν μορφή που να τις κάνει χρήσιμες και πέρα από το συγκεκριμένο πρόγραμμα.[7]

Ας πάρουμε τη *natProduct()*. Για αυτήν η προϋπόθεση είναι:

$$0 < m \leq n$$

Θα μας δίνει πάντοτε ως αποτέλεσμα έναν θετικό ακέραιο. Για να επιστρέψουμε μήνυμα λάθους

- μπορούμε να βάλουμε μία ακόμη παράμετρο, οπότε θα την κάνουμε (σύμφωνα με τις συνήθειές μας) `void`,

```
// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
void natProduct( int m, int n,
                 unsigned long int& fv, int& err )
{
   if ( m <= 0 || n < m )
      err = 1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
      err = 0;
   }
} // natProduct
```

- μπορούμε όμως να επιστρέφουμε και κάποια αρνητική τιμή:

```
long int natProduct( int m, int n )
{
   long int fv;

   if ( m <= 0 || n < m )
      fv = -1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
   }
   return fv;
} // natProduct
```

Πώς γίνεται τώρα η *factorial()*; Για την πρώτη περίπτωση θα έχουμε:

```
void factorial( int a,
                unsigned long int& fv, int& err )
{
   if ( a == 0 ) {  fv = 1;  err = 0;  }
```

[7] Η τελευταία απαίτηση μας λέει απλούστατα ότι δεν μπορούμε να στηριχθούμε σε έναν έλεγχο στη **main** και να «καθαρίσουμε» για τα πάντα.

```
            else natProduct( 1, a, fv, err );
} // factorial
```

ενώ για τη δεύτερη περίπτωση παραμένει σχεδόν όπως ήταν:

```
long int factorial( int a )
{
   return ( (a == 0) ? 1 : natProduct(1, a) );
} // factorial
```

Το μόνο που άλλαξε είναι ο τύπος του αποτελέσματος –που έγινε `long int`– για να επιτρέπει την επιστροφή της τιμής "`-1`" σε περίπτωση λάθους.

Η εντυπωσιακή αλλαγή γίνεται στην *comb()*:

```
// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
void comb( int m, int n,
           unsigned long int& fv, int& err )
{
   unsigned long int v1, v2;

   if ( n < m-n )
   {
      factorial( n, v1, err );
      if ( err == 0 )
      {
         natProduct( m-n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
   else
   {
      factorial( m-n, v1, err );
      if ( err == 0 )
      {
         natProduct( n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
} // comb
```

Φρίκη! Υπολογισμοί και διαχείριση σφαλμάτων έγιναν «μαλλιά κουβάρια»! Σύγκρινε αυτήν την εκδοχή με την αρχική... Ίδια είναι η κατάσταση και στην περίπτωση που η συνάρτηση επιστρέφει τιμή "`-1`" (άσκηση για σένα!)

Στη **main** θα έχουμε:

```
int main()
{
   int m, n;
   unsigned long int s;
   int  err;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   comb( m, n, s, err );
   if ( err == 0 )
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << s << endl;
   else
      cout << " Λάθος δεδομένα" << endl;
} // main
```

Και στις δύο περιπτώσεις, έχουμε χάσει την ευκολία και την κομψότητα της κλήσης μιας συνάρτησης με τύπο.

Να συγκρίνουμε αυτό το πρόγραμμα και την εκτέλεσή του με αυτό που γράψαμε στην §14.9.1:

1. Το πρόγραμμα με την εξαίρεση είναι πολύ πιο απλό και πιο κατανοητό. Η διαφορά έγκειται στη δυνατότητα που έχουμε να καλούμε τις συναρτήσεις (με τύπο) μέσα σε παραστάσεις.
2. Στο πρώτο πρόγραμμα το ξετύλιγμα της στοίβας γίνεται αυτομάτως και πολύ απλά. Εδώ, γίνεται βήμα-προς-βήμα αφού κάθε φορά ελέγξουμε την τιμή της *err*.
3. Το δεύτερο πρόγραμμα έχει γεμίσει με εντολές για τη διαχείριση κωδικών λάθους. Πιο χαρακτηριστική είναι η περίπτωση της *comb()*.

Βλέπουμε λοιπόν δύο σημαντικά πλεονεκτήματα του μηχανισμού διαχείρισης εξαιρέσεων που προσφέρει η C++ (αλλά και άλλες γλώσσες προγραμματισμού):

- Το αυτόματο ξετύλιγμα της στοίβας και η δυνατότητα χρήσης συναρτησεων με τύπο.
- Ο συστηματικός *διαχωρισμός των εντολών διαχείρισης εξαιρετικών συμβάντων* από το υπόλοιπο πρόγραμμα.

Στη συνέχεια θα δούμε και άλλα.

24.3 Συμπληρώματα στην «Ιστορία με Εξαιρέσεις»

Θα δώσουμε τώρα μερικά συμπληρώματα στην «Ιστορία με Εξαιρέσεις» που είδαμε στην §14.9.1. Θα ξεκινήσουμε όμως με έναν ορισμό που αντιγράφουμε από το Java Tutorial:[8] «*Ο όρος εξαίρεση είναι συντομογραφία της φράσης "εξαιρετικό συμβάν."*

♦ ***Μια εξαίρεση (exception) είναι ένα συμβάν –κατά τη διάρκεια της εκτέλεσης ενός προγράμματος– που διακόπτει την ομαλή ροή των εντολών του προγράμματος.»***

Όταν σε κάποιο σημείο μιας συνάρτησης βρούμε να μην ισχύει κάποια συνθήκη που είναι αναγκαία, **ρίχνουμε** (throw) ή **εγείρουμε** (raise) μιαν εξαίρεση:

"`throw`", *αντικείμενο εξαίρεσης* ;

π.χ.:

```
throw -1;      throw n;      throw x;
```

Συνήθως χρησιμοποιούμε τον όρο *εξαίρεση* αντί για *αντικείμενο εξαίρεσης*.

Τα χαρακτηριστικά μιας εξαίρεσης είναι:

- ο τύπος και
- *η τιμή*

του αντικειμένου της.

Στη συνάρτηση *v()* που είδαμε στην §14.9 έχουμε την εντολή "`throw x;`". Τι θα γίνει αν εκτελεσθεί αυτή η εντολή; Σύμφωνα με αυτά που είδαμε στην §14.9.1, «*Τα σχετικά με την κλήση της*» *v()* «*φεύγουν από τη στοίβα και επιστρέφουμε*» στη συνάρτηση που κάλεσε τη *v()*. Και η *x*; Αυτή μας χρειάζεται, αφού είναι το αντικείμενο της εξαίρεσης (και θα πρέπει να γίνει τιμή της παραμετρου της **catch** που θα συλλάβει την εξαίρεση)!

Αυτό που γίνεται είναι το εξής: όταν εκτελεσθεί η **throw** το αντικείμενο της εξαίρεσης αντιγράφεται σε «σίγουρη» θέση όπου ζει μέχρι να τελειώσει η τελευταία **catch** που θα τη συλλάβει (και δεν θα την ξαναρίξει με "`throw;`"). Μετά την εκτέλεση αυτής της **catch** θα κληθεί ο καταστροφέας να καταστρέψει την εξαίρεση.

Αν το αντικείμενο της εξαίρεσης είναι μια σταθερά, όπως το "`-1`" που ρίχνει η *natProduct()*, ή ένα αντικείμενο χωρίς όνομα, όπως "`StudentXptn("Student", StudentXptn::noMemory)`", ο μεταγλωττιστής *μπορεί να κανονίσει* να μη χρειαστεί αντιγραφή (φυλάγοντας καταλλήλως το αντικείμενο της εξαίρεσης εξ αρχής).

Σε κάθε περίπτωση, τα παραπάνω βάζουν την εξής υποχρέωση στον προγραμματιστή που γράφει δικές του κλάσεις εξαιρέσεων:

---

[8] `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`

- ***Μια κλάση εξαιρέσεων θα πρέπει να έχει σωστό δημιουργό αντιγραφής και σωστό καταστροφέα.***

Όσο η εξαίρεση «ταξιδεύει», κατά το «*ξετύλιγμα της στοίβας*», καλούνται καταστροφείς για να καταστρέψουν τα τοπικά αντικείμενα στις συναρτήσεις που κλείνουν. Όταν εκτελείται κάποιος από αυτούς τους καταστροφείς ή –κάποια συνάρτηση που καλείται από κάποιον καταστροφέα– μπορούμε να ελέγξουμε αν υπάρχει εξαίρεση που δεν έχει συλληφθεί ελέγχοντας την τιμή που επιστρέφει η συνάρτηση (*std*::)*uncaught_exception()*.

Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται αυτομάτως η (*std*::)*terminate()* (που καλεί την (*std*::)*abort()*) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_terminate()*– να αντικαταστήσεις την *terminate()* με δική σου.

Ας πούμε τώρα ότι είχες *προδιαγραφή εξαιρέσεων*:

```
unsigned long int natProduct( int m, int n ) throw( char* )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// . . .
} // natProduct
```

και εκτελείται η "`throw -1`". Στην περίπτωση αυτήν –και κάθε φορά που ρίχνεται εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση (*std*::)*unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Όπως στην περίπτωση της *terminate()*, η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_unexpected()*– να αντικαταστήσεις την (*std*::)*unexpected()* με δική σου.

24.4 Οι Συναρτήσεις Διαχείρισης Εξαιρέσεων

Θα δούμε τώρα τις συναρτήσεις που δίνει η C++[9] για τη διαχείριση εξαιρέσεων. Αλλά να τονίσουμε από την αρχή το εξής: *Αν σχεδιάσεις καλά το προγράμμά σου, αυτές οι συναρτησεις δεν θα σου χρειαστούν*. Αργότερα όμως –όταν οι τροποποιήσεις και οι προσαρμογές στις απαιτήσεις που αλλάζουν– δεν θα καλύπτονται από τις προβλέψεις του αρχικού σχεδίου, αυτές οι συναρτήσεις μπορεί να σε βοηθήσουν να κάνεις διάφορα «μπαλώματα».

24.4.1 Η Συνάρτηση `std::set_terminate()`

Λέγαμε στην προηγούμενη παράγραφο: «*... καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (std::)set_terminate()– να αντικαταστήσεις την terminate() με δική σου.*»

Φαντάσου τώρα ότι η *terminate()*, για την οποίαν τα λέμε στη συνέχεια, είναι κάτι σαν:

```
void terminate()
{
   terminate_handler pDth( &dth );
// . . .
   (*pDth)();
}
```

όπου

```
void dth()
{
// . . .
```

[9] Αναφερόμαστε στη C++03. Η C++11 δίνει και άλλες...

```
   abort();
}
```

και

```
typedef void (*terminate_handler)();
```

Δηλαδή, ο *terminate_handler* είναι τύπος βέλους προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την παράμετρο της *atexit()*).

Όπως καταλαβαίνεις, η *terminate()* καλεί την *abort()* *εμμέσως*, μέσω της *dth()*.

Η *set_terminate()* σου επιτρέπει να αλλάξεις τη συνάρτηση που καλεί η *terminate()* και –στο παράδειγμά μας– δείχνει το *pDth*. Για να τη χρησιμοποιήσεις θα θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <exception>`". Στο `exception` μπορείς να δεις την επικεφαλίδα της:

```
terminate_handler set_terminate(terminate_handler f) throw();
```

Ας πούμε ότι γράφουμε τη:

```
void myTerminateHandler()
{
   logFl << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
         << "Φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler
```

και μέσα στο πρόγραμμά μας δίνουμε:

```
   terminate_handler oldTH;
   oldTH = set_terminate( &myTerminateHandler );
```

Τι θα γίνει;

- Ο *pDth* της *terminate()* θα πάρει τιμή "`&myTerminateHandler`", τη διεύθυνση του νέου χειριστή τερματισμού.
- Η *oldTH* θα πάρει ως τιμή αυτήν που είχε πριν η *pDth*, θα δείχνει τον παλιό χειριστή τερματισμού.

Έτσι δουλεύει η *set_terminate()*.[10] Η τιμή που επιστρέφει η συνάρτηση μας είναι συνήθως άχρηστη (εκτός από την περίπτωση που θα θελήσεις να ξαναεγκαταστήσεις αργότερα τον αρχικό χειριστή) και το πιθανότερο είναι να δίνεις απλώς:

```
   set_terminate( &myTerminateHandler );
```

Δες ένα πρόγραμμα που δοκιμάζει τα παραπάνω:

```
#include <fstream>
#include <exception>
#include <cstdlib>

using namespace std;

ofstream logFl( "log.txt" );

void myTerminateHandler()
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   set_terminate( &myTerminateHandler );
   throw -1;
}
```

Γιατί βάλαμε τη συνάρτησή μας να γράφει σε αρχείο και όχι στην οθόνη; Για να κάνουμε φανερό ότι αν θέλουμε κάτι τέτοιο θα πρέπει να το κάνουμε με ένα καθολικό αντικείμενο, όπως είναι το ρεύμα *logFl*, αφού η συνάρτησή μας δεν (μπορεί να) έχει παραμέτρους.

---

[10] Τώρα καταλαβαίνεις γιατί λέγαμε «φαντάσου»: εδώ φαίνεται ότι έρχεται η *set_terminate()* να αλλάξει την τιμή μιας τοπικής μεταβλητής (*pDth*) της *terminate()*…

Αν θελήσεις να γράψεις δική σου συνάρτηση για τον χειρισμό του τερματισμού πρόσεξε το εξής:

- ***Η συνάρτηση χειρισμού τερματισμού δεν θα πρέπει να επιστρέφει ως συνήθως με "`return`" αλλά θα πρέπει να τερματίζει την εκτέλεση του προγράμματος, κατά προτίμηση με κλήση της abort().***

  Την *abort()* καλεί και η συνάρτηση που βάζει αρχικώς ο μεταγλωττιστής.

24.4.2 Η Συνάρτηση `std::terminate()`

Όπως λέγαμε: «*Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί, καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος.*» Αυτό το βλέπεις και στο παράδειγμα που δώσαμε για τη *set_terminate()*: Στη `main` υπάρχει η "`throw -1`" που θα εκτελεσθεί σίγουρα αλλά δεν υπάρχει **catch** για να τη συλλάβει.

Στη συνέχεια βλέπεις πολλές περιπτώσεις που η *terminate()* καλείται αυτομάτως. Πάντως μπορείς να βάλεις και εντολή κλήσης της στο πρόγραμμά σου αρκεί να βάλεις μια "`#include <exception>`":

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

int main()
{
   set_terminate( &myTerminateHandler );
   terminate();
}
```

Αποτέλεσμα:[11]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

Στο `exception` μπορείς να δεις δήλωση που μοιάζει με:

```
void terminate();
```

Στη συνέχεια παραθέτουμε όλες τις περιπτώσεις που καλείται αυτομάτως η *terminate()* για να διακόψει την εκτέλεση του προγράμματός σου. Πάντοτε η κλήση της γίνεται πριν από το «ξετύλιγμα της στοίβας» εκτός φυσικά από την δεύτερη περίπτωση (δεν βρέθηκε **catch** να συλλάβει την εξαίρεση). Η *std::terminate()* καλείται:[12]

- Όταν ο μηχανισμός διαχείρισης εξαιρέσεων, αφού δημιουργήσει το αντικείμενο της εξαίρεσης και πριν αυτό συλληφθεί από κάποια **catch**, καλέσει κάποια συνάρτηση που ρίχνει εξαίρεση. Ας πούμε ότι έχουμε:

```
struct Xptn
{
```

[11] Borland C++ v.5.5.

[12] Αντιγράφουμε από το πρότυπο…

```
   int c;
   Xptn( int ac=0 )  {  c = ac;  }
   Xptn( const Xptn& rhs ) {  throw -1;  }
}; // Xptn
```

και

```
int main()
{
   Xptn x;
   set_terminate( &myTerminateHandler );
   try {
      throw x;
   }
   catch( Xptn& )
   {  cout << "in catch( Xptn& )" << endl;  }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Αποτέλεσμα:[13]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Το αντικείμενο που ρίχνει η "`throw x`" είναι έτοιμο. Όταν γίνει προσπάθεια αντιγραφής σε «σίγουρη» θέση –όπως μάθαμε στην §24.3– καλείται ο δημιουργός αντιγραφής. Αυτός ρίχνει άλλη εξαίρεση (`throw -1`) και έτσι καλείται η *terminate()*.

- Όταν έχει ριχτεί μια εξαίρεση, το ξετύλιγμα της στοίβας έχει φτάσει στη **main** και η εξαίρεση δεν έχει συλληφθεί από κάποια **catch**. Ας πούμε ότι έχουμε την *Xptn* και τη *myTerminateHandler()* που είχαμε παραπάνω και ακόμη:

```
void f1()
{  throw Xptn(-1);  }

void f2()
{  f1();  }

int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f2();
   }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Το πρόγραμμα θα μας δώσει το ίδιο αποτέλεσμα που πήραμε και από το προηγούμενο παράδειγμα. Πώς εξηγείται; Η **main** καλεί την *f2()* και αυτή την *f1()* που ρίχνει εξαίρεση "`Xptn(-1)`". Με το «ξετύλιγμα της στοίβας» επιστρέφουμε στην *f2()* και στη συνέχεια στη **main**. Η εξαίρεση δεν συλλαμβάνεται από την "`catch(int&)`" και αφού δεν υπάρχει "`catch(Xptn&)`" ούτε "`catch(...)`" καλείται η *terminate()*.

- Όταν κατά τη διάρκεια του ξετυλίγματος κληθεί κάποιος καταστροφέας ο οποίος ρίχνει μιαν εξαίρεση. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  }
   ~C()  {  throw 0;  }
};
```

[13] gcc.

```
void f()
{  C lc( 7 );
   throw -1;  }
```

και

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f();
   }
   catch( int& )
   {  cout << "int exception" << endl;  }
}
```

Μετά την εκτέλεση της "`throw -1`" καλείται ο καταστροφέας της C για να καταστρέψει το *lc*. Αλλά αυτός ρίχνει εξαίρεση και έτσι καλείται η *terminate()*· το αποτέλεσμα είναι αυτό που είδαμε στα άλλα παραδείγματα.

- Όταν ριχτεί εξαίρεση κατά τη δημιουργία ή την καταστροφή ενός στατικού αντικειμενου. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  throw -1;  }
};

C gc;
```

Το *gc* είναι ένα καθολικό στατικό αντικείμενο. Με τη εκτέλεση της "`throw -1`" κατά τη δημιουργία του καλείται η *terminate()*.[14] Στην περίπτωση που έχουμε:

```
void f()
{
   static C c( 7 );
}

int main()
{
   set_terminate( &myTerminateHandler );
   f();
}
```

και πάλι, κατά τη δημιουργία του *c*, θα έχουμε κλήση της *terminate()* αλλά τώρα θα δούμε και τους στίχους από το γνωστό άσμα.

- Όταν κατά την εκτέλεση συνάρτησης που έχουμε καταχωρίσει με την *atexit()* ριχτεί εξαίρεση. Αλλάζουμε λίγο τη *bye()* που είδαμε στην §24.1.2:

```
void bye()
{  cout << "Bye bye!" << endl;
   throw -1;  }
```

και δοκιμάζουμε τη

```
int main()
{
   set_terminate( &myTerminateHandler );
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   exit( 0 );
}
```

Αποτέλεσμα:

```
Bye bye!
```

[14] Πριν αρχίσει η εκτέλεση της **main** και επομένως χωρίς να εκτελεσθεί η "`set_terminate( &myTerminateHandler )`".

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

- Όταν γίνει απόπειρα εκτέλεσης μιας "`throw;`" (που ξαναρίχνει την εξαίρεση που χειριζόμαστε) ενώ δεν γίνεται διαχείριση κάποιας εξαίρεσης, π.χ.:

```
int main()
{
   set_terminate( &myTerminateHandler );
   throw;
}
```

- Από την *unexpected()*, όπως θα δούμε παρακάτω.

Από τα παραπάνω βγαίνουν και δύο κανόνες προγραμματιστικής πρακτικής· τον πρώτο τον έχουμε ήδη διατυπώσει:

♦ ***Οι καταστροφείς δεν επιτρέπεται να ρίχνουν εξαιρέσεις.***

Θα πεις «δεν μπορώ να φανταστώ περίπτωση που θα έβαζα μια εντολή `throw` σε έναν καταστροφέα.» Σωστό βέβαια, αλλά πρόσεχε αν ο καταστροφέας καλεί άλλες συναρτήσεις. Στην περίπτωση αυτή, ο καταστροφέας θα πρέπει να συλλαμβάνει όλες τις εξαιρέσεις και να σταματάει την μετάδοσή τους.

Τον δεύτερο τον τηρούμε παρ' όλο που δεν τον διατυπώσαμε:

♦ ***Αν γράφεις δικές σου κλάσεις εξαιρέσεων καλό είναι να μη ρίχνονται εξαιρέσεις όχι μόνο από τους καταστροφείς τους αλλά και από τους δημιουργούς τους.***

24.4.3 * Προδιαγραφές Εξαιρέσεων και Σχετικές Συναρτήσεις

Η (*std*::)*unexpected()* καλείται αυτομάτως όταν μια συνάρτηση –που έχει προδιαγραφή εξαιρέσεων– ρίξει εξαίρεση που ο τύπος της δεν υπάρχει στη λίστα τύπων της προδιαγραφής. Η (*std*::)*set_unexpected()* σου δίνει τη δυνατότητα να αλλάζεις την *unexpected()*.

Οι προδιαγραφές εξαιρέσεων έχουν ταλαιπωρήσει αρκετά τους προγραμματιστές που δοκίμασαν να τις χρησιμοποιήσουν. Οι γνώμες ενάντια στο εργαλείο αυτό ήταν πολλές και στο C++11 αποθαρρύνεται η χρήση τους εκτός από την περίπτωση "`throw()`" (η συνάρτηση δεν ρίχνει εξαίρεση). Έτσι λοιπόν και εμείς δίνουμε τη συμβουλή:

♦ ***Μη χρησιμοποιείς προδιαγραφές εξαιρέσεων εκτός από τη "`throw()`" και κατά συνέπεια τις συναρτησεις* unexpected() *και* set_unexpected().**

Πάντως στη συνέχεια θα δούμε εν συντομία τις δύο συναρτήσεις και τη χρήση της κλάσης (*std*::)*bad_exception*.

Ας πούμε ότι έχουμε το πρόγραμμα:

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

unsigned long int natProduct( int m, int n ) throw( char* )
{
   throw -1;
} // natProduct
```

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
}
```

Αποτέλεσμα:

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Τι έγινε; Η *natProduct()* έριξε εξαίρεση με αντικείμενο "`-1`" (`int`) εκτός προδιαγραφών εξαιρέσεων. Κλήθηκε η *unexpected()* και αυτή με τη σειρά της κάλεσε την *terminate()*.

Τώρα κάνουμε τις εξής αλλαγές:

- Γράφουμε την:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   throw;
}
```

- Αλλάζουμε την προδιαγραφή εξαιρέσεων της *natProduct()*:

```
unsigned long int natProduct( int m, int n ) throw( char*, bad_exception )
{
   throw -1;
} // natProduct
```

- Τέλος, αλλάζουμε και τη **main**:

```
int main()
{
   set_terminate( &myTerminateHandler );
   set_unexpected( &myUnexpected );      // <<<<<<<<<<<<<<
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
   catch( bad_exception& )               // <<<<<<<<<<<<<<
   {  cout << "catching bad_exception" << endl;  }
}
```

Αποτέλεσμα:

```
unexpected; ασ' το σε μένα...
catching bad_exception
```

Βάζοντας στη λίστα τύπων εξαιρέσεων τον (*std*::)*bad_exception* και γράφοντας τη δική μας *myUnexpected()*, γλυτώσαμε από την *terminate()*!

24.4.4 * Η Συνάρτηση `std::uncaught_exception()`

Η επικεφαλίδα της συνάρτησης, όπως μπορείς να τη δεις στο `exception`, είναι:

```
bool uncaught_exception() throw();
```

Η συνάρτηση αυτή επιστρέφει τιμή "`true`" αν, όταν κληθεί, υπάρχει εξαίρεση που δεν έχει συλληφθεί.

Ας δούμε ένα παράδειγμα χρήσης. Η *myUnexpected()* που γράψαμε παραπάνω έχει το εξής πρόβλημα: Ενώ μπορεί να κληθεί κατ' ευθείαν από το πρόγραμμά μας, σε τέτοια περίπτωση, αν δεν υπάρχει κάποια εξαίρεση «στον αέρα», η "`throw;`" θα προκαλέσει την (αυτοματη) κλήση της *terminate()*. Η σωστή *myUnexpected()* είναι:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   if ( uncaught_exception() )
      throw;
}
```

Πού αλλού θα μπορούσε να χρησιμοποιηθεί αυτή η συνάρτηση; Για να την απαντησουμε θα πρέπει να σκεφτούμε ποιες συναρτήσεις ενεργοποιούνται όταν επιστρέφει "`true`" αυτή η συνάρτηση, δηλαδή από τη στιγμή που ρίχτηκε μια εξαίρεση μέχρι να συλληφθεί. Τι γίνεται στο διάστημα αυτό; «Κλείνουν» διάφορες συναρτήσεις και καλούνται καταστροφείς για να καταστρέψουν τα αντικείμενα που ζούσαν μέσα σε αυτές. Επομένως;... Αυτό που κατάλαβες! Μπορείς να τη χρησιμοποιήσεις μόνο μέσα σε καταστροφείς (ή σε συναρτήσεις που αυτοί καλούν). Και πώς θα τη χρησιμοποιήσεις; Θα γράφεις κάτι σαν:

```
   if ( uncaught_exception() )
   {  E1  }
   else
   {  E2  }
```

που σημαίνει: αν έχουμε εξαίρεση «στον αέρα» θα καταστρέψεις το αντικείμενο με τις *E1* αλλιώς θα το καταστρέψεις με τις *E2* (που μπορεί να ρίχνουν και καμιά εξαίρεση).

Ένας προγραμματιστής που σέβεται τον εαυτό του δεν θα γράψει κάτι τέτοιο ακόμη και στην πιο απελπισμένη προσπάθεια να «μπαλώσει» ένα προβληματικό πρόγραμμα.

24.5 Συναρτησιακή Ομάδα try

Στην §23.3 γράψαμε τον δημιουργό της *DateTime* ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
// . . .
}; // DateTime::DateTime
```

Αν ο δημιουργός της *Date* ρίξει εξαίρεση εμείς θα πάρουμε το μήνυμα ότι κάποιο πρόβλημα βρήκε ο δημιουργός της *Date*· αυτό όμως είναι ψέμα ή –τουλάχιστον– μισή αλήθεια: Το πρόβλημα βρήκε ο δημιουργός της *Date όταν κλήθηκε από τον δημιουργό της DateTime*.

Τι μπορούμε να κάνουμε για να το διορθώσουμε; Να πιάσουμε την εξαίρεση *DateXptn* από τον δημιουργό της *Date* και να ρίξουμε νέα εξαίρεση *DateTimeXptn*. Αυτό μπορεί να γίνει ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}
catch( DateTimeXptn& x )
```

```
{  throw;  }
catch( DateXptn& x )
{
   throw DateTimeXptn( "DateTime", x.errorCode,
                       x.errVal1, x.errVal2, x.errVal3 );
} // DateTime::DateTime
```

Τι έχουμε εδώ;

- Η ομάδα της **try** ταυτίζεται με το σώμα της συνάρτησης· από εδώ και το όνομα **συναρτησιακή ομάδα try** (functional try-block). Οι **catch** ανήκουν στη συνάρτηση παρ' όλο που φαίνεταο να είναι έξω από αυτήν.
- Η λίστα εκκίνησης ελέγχεται από την **try**, παρ' όλο που βρίσκεται πριν από το "`{`".
- Η πρώτη **catch** πιάνει τις εξαιρέσεις που ρίχνονται από τον ίδιο τον δημιουργό και τις ξαναρίχνει χωρίς άλλη ενέργεια ("`throw;`").
- Η δεύτερη **catch** πιάνει οποιαδήποτε εξαίρεση ριχτεί από τη λίστα εκκίνησης –στην περίπτωσή μας από την κλήση του δημιουργού της βασικής– και την ξαναρίχνει αφού τη μετατρέψει σε *DateTimeXptn*.

Αμφότερες οι ομάδες **catch** τελειώνουν με `throw`. Σύμπτωση; Όχι!

♦ ***Αν έχεις συναρτησιακή ομάδα try σε δημιουργό όλες οι ομάδες catch που αντιστοιχούν σε αυτήν πρέπει να τελειώνουν με `throw`. Αν δεν υπάρχει τοτε ο μεταγλωττιστής θα βάλει ερήμην σου μια "`throw;`".***

Φυσικά, αυτό δεν είναι περίεργο: το ότι ήλθε κάποια εξαίρεση σημαίνει ότι δεν είναι δυνατή η δημιουργία κάποιου μέλους του αντικειμένου (ή του υποαντικειμένου της βασικής κλάσης). Επομένως δεν είναι δυνατόν να δημιουργηθεί το αντικείμενο!

Αυτά που ισχύουν για συναρτησιακή ομάδα **try** *σε δημιουργό ισχύουν και για τον καταστροφέα.* Αφού όμως έχουμε εξηγήσει ότι από τον καταστροφέα δεν επιτρέπεται να ξεφεύγουν εξαιρέσεις, η δυνατότητα να βάλουμε συναρτησιακή ομάδα **try** σε καταστροφέα μας είναι τελείως άχρηστη!

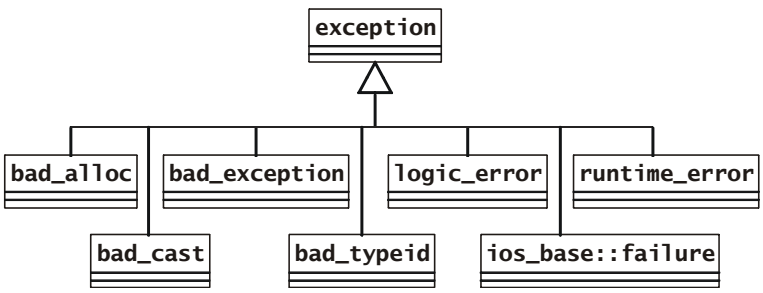
Μπορείς να βάλεις συναρτησιακή ομάδα **try** σε οποιαδήποτε συνάρτηση εκτός από τη **main**. Στην περίπτωση αυτή έχουμε μια διαφορά από αυτά που ισχύουν για δημιουργούς και καταστροφείς: *δεν υπάρχει υποχρέωση για* `throw` *στις αντίστοιχες ομάδες* **catch**. Αλλά και αυτή η δυνατότητα μας είναι άχρηστη αφού

Η
```
void f( . . . )
try {
// . . .
}
catch( . . . )
{
// . . .
}
```
είναι ισοδύναμη με
```
void f( . . . )
{
   try
   {
   // . . .
   }
   catch( . . . )
   {
   // . . .
   }
}
```

24.6 Οι Τύποι Εξαιρέσεων της C++

Η C++ έχει μερικές κλάσεις εξαιρέσεων για χρήση στις βιβλιοθήκες της· οι κλάσεις αυτές είναι διαθέσιμες και στους προγραμματιστές. Σύμφωνα με το πρότυπο της γλώσσας, η κλάση αυτή, που δηλώνεται στο `exception`, πρέπει να είναι:

```
class exception
{
public:
   exception() throw();
   exception( const exception& ) throw();
   exception& operator=( const exception& ) throw();
```



Σχ. 24-1 Η *exception* και οι επτά κλάσεις εξαιρέσεων που την κληρονομούν. Οι *logic_error* και *runtime_error* έχουν δικές τους κλάσεις-κληρονόμους.

```
   virtual ~exception() throw();
   virtual const char* what() const throw();
};
```

Όπως βλέπεις, θα πρέπει να έχει:

- ερήμην δημιουργό,
- δημιουργό αντιγραφής,
- τελεστή εκχώρησης,
- καταστροφέα (`virtual`) και
- μια μέθοδο (`virtual`), τη *what()*, που επιστρέφει ένα κείμενο.

Τα πάντα έχουν το χαρακτηριστικό "`throw()`": *δεν ρίχνουν εξαιρέσεις!* Στην §16.6.1 λέγαμε ότι, στις δικές μας κλάσεις εξαιρέσεων, έπρεπε να βάλουμε "`char funcName[100]`" και όχι "`string funcName`" για να έχουμε τη σιγουριά ότι δεν θα ριχτεί εξαίρεση όταν καλείται ο δημιουργός για να δημιουργήσει ή να αντιγράψει ένα αντικείμενο εξαίρεσης. Το πρότυπο λέει: «*Για παράδειγμα, αν το αντικείμενο [εξαίρεσης] είναι κλάσης με δημιουργό αντιγραφής θα κληθεί η* `std::terminate()` *αν η εκτέλεση του δημιουργού αντιγραφής διακοπεί με εξαίρεση όταν εκτελείται μια* `throw`.»

Τι είναι η *what()*; Μια μέθοδος που επιστρέφει ένα κείμενο-μήνυμα. Για παράδειγμα στη Borland C++ v.5.5 το μήνυμα λέει:

```
no named exception thrown
```

Αυτό που είδαμε παραπάνω δεν είναι δήλωση κλάσης· μπορείς να το θεωρήσεις ως διεπαφή.[15] Η κλάση *exception* δεν χρησιμοποιείται στα προγράμματα. Χρησιμοποιούνται οι παράγωγές της που τις βλέπεις στο Σχ. 24-1.

Η πρώτη παράγωγη που ήδη χρησιμοποιούμε είναι η (*std*::)*bad_alloc*. Το πρότυπο λέει:

```
class bad_alloc : public exception
{
public:
   bad_alloc() throw();
   bad_alloc( const bad_alloc& ) throw();
   bad_alloc& operator=( const bad_alloc& ) throw();
   virtual ~bad_alloc() throw();
   virtual const char* what() const throw();
};
```

Όπως βλέπεις, είναι ολόιδια με τη βασική. Βέβαια, εδώ η *what()* βγάζει άλλο μήνυμα (BC++ v.5.5):

```
bad alloc exception thrown
```

Στη C++ που χρησιμοποιείς θα βρεις τη δήλωσή της στο `new`.

Παρόμοια ισχύουν και για τις άλλες παράγωγες κλάσεις, εκτός από την `ios_base::failure`.

---

[15] Αν ψάξεις στο `exception` μπορεί να βρεις πώς το ολοκληρώνει ως κλάση η C++ που χρησιμοποιείς.

Γνωρίσαμε τη (*std*::)*bad_cast* στην §23.13.1 και στην §Prj06.4. Η δήλωσή της στο `typeinfo`.

Στην §24.4.3 είδαμε την (*std*::) *bad_exception*. Η δήλωσή της στο `exception`.

Η (*std*::)*bad_typeid* ρίχνεται από την *typeid* αν της δώσεις όρισμα (αποπαραπομπή σε) βέλος "`0`". Για να τη χρησιμοποιήσεις πρέπει να βάλεις "`#include <typeinfo>`". Για παράδειγμα, οι:

```
   Student* p( 0 );
   try {
      cout << typeid(*p).name() << endl;
// . . .
   }
   catch ( bad_typeid& x )
   {
      cout << "bad_typeid caught" << endl;
      cout << x.what() << endl;
   }
```

θα δώσουν (g++):

```
bad_typeid caught
St10bad_typeid
```

Εξαιρέσεις κλάσης (*std*::)*ios_base*::*failure* ρίχνονται όταν αποτύχουν μερικές πράξεις διαχείρισης ρευμάτων.[16] Ο ορισμός της κλάσης έχει την εξής διαφορά από τις άλλες κλάσεις που είδαμε: έχει έναν ακόμη δημιουργό

```
   explicit failure( const string& msg );
```

με τον οποίον εισάγουμε (τιμή του *msg*) το κείμενο που μας δίνει η *what()*.

24.6.1 Η Κλάση *logic_error* και οι Παράγωγές της

Η *logic_error* ορίζεται στο `stdexcept`:

```
class logic_error : public exception
{
public:
   explicit logic_error( const string& what_arg );
};
```

Δηλαδή, εκτός από αυτά που έχει η βασική υπάρχει ένας επιπλέον δημιουργός όπως αυτός που είδαμε στην *ios_base*::*failure*.

Όπως λέει το πρότυπο, «*η κλάση logic_error ορίζει τον τύπο των αντικειμένων που ρίχνονται ως εξαιρέσεις για να αναφέρουν σφάλματα υποθετικώς ανιχνεύσιμα πριν από την εκτέλεση του προγράμματος, όπως παραβιάσεις λογικών προϋποθέσεων ή αναλλοίωτων κλάσεων.*»

24.6.1.1 Η Κλάση *domain_error*

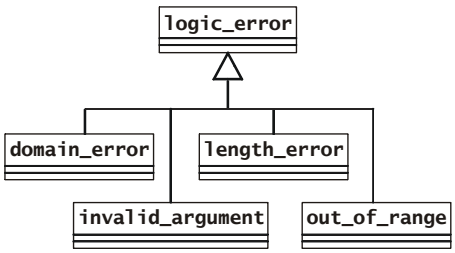
Η πρώτη παράγωγη κλάση είναι η:

```
class domain_error : public logic_error
{
public:
   explicit domain_error( const string& what_arg );
};
```

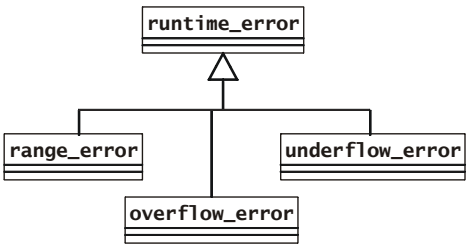
Τη ρίχνουμε όταν μια συνάρτηση καλείται με ορίσματα έξω από το **πεδίο ορισμού** (domain of definition) της. Θα μπορούσαμε να γράψουμε τη *factorial()* που είδαμε στις *Εισαγωγικές Παρατηρήσεις* ως εξής:

---

[16] Στο C++11 η κλάση αυτή είναι παράγωγη της *system_error* που, με τη σειρά της, είναι παράγωγη της *runtime_error*.



Σχ. 24-2 Η *logic_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.



Σχ. 24-3 Η *runtime_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.

```
unsigned long int factorial( int a )
{
   if ( a < 0 )
   {
      ostringstream ssout;
      ssout << "η factorial κλήθηκε με όρισμα " << a;
      throw domain_error( ssout.str() );
   }
   unsigned long int fv( 1 );
   for ( int k(1);  k <= a;  ++k )  fv *= k;
   return fv;
} // factorial
```

Δημιουργούμε το αντικείμενο της εξαίρεσης με τον δημιουργό της κλάσης συναρμολογώντας το μήνυμα με ένα ρεύμα κλάσης *ostringstream* (§10.12).

Παράδειγμα χρήσης:

```
#include <iostream>
#include <stdexcept>  // για τη domain_error
#include <sstream>    // για το ostringstream

using namespace std;

unsigned long int factorial( int a )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main ()
{
   try
   {
      cout << factorial( 7 ) << endl;
      cout << factorial( -7 ) << endl;
   }
   catch ( domain_error& x )
   {
      cout << "domain error: " << x.what() << endl;
   }
}
```

Αποτέλεσμα:

```
5040
domain error: η factorial κλήθηκε με όρισμα -7
```

24.6.1.2 Η Κλάση *invalid_argument*

Ορίζεται στο `stdexcept` ως:

```
class invalid_argument : public logic_error
{
public:
   explicit invalid_argument( const string& what_arg );
};
```

Ένα «προβληματικό» όρισμα συνάρτησης συνήθως έχει τιμή εκτός πεδίου ορισμού και το αντιμετωπίζουμε με "`domain_error`". Αν έχει άλλο πρόβλημα χρησιμοποιούμε την "`invalid_argument`".

24.6.1.3 Η Κλάση *length_error*

Ορίζεται στο `stdexcept` ως:

```
class length_error : public logic_error
{
public:
   explicit length_error( const string& what_arg );
};
```

Θα μπορούσε να ονομάζεται και "`size_error`". Τη ρίχνουμε όταν το «μέγεθος» (ό,τι και αν είναι αυτό) κάποιου αντικειμένου υπερβαίνει κάποιο όριο που βάζει η υλοποίηση (ή το πρόγραμμα εφαρμογής). Για παράδειγμα, αν έχουμε δηλώσει:

```
   string s1;
```

οι εντολές:

```
   try {
      s1.reserve( s1.max_size()+1 );
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):[17]

```
std::length_error
invalid string size parameter in function: basic_string::reserve(size_t)
 size: -14 is greater than maximum size: -15
```

24.6.1.4 Η Κλάση *out_of_range*

Ορίζεται στο `stdexcept` ως:

```
class out_of_range : public logic_error
{
public:
   explicit out_of_range( const string& what_arg );
};
```

Θα τη δούμε με ένα παράδειγμα· αν έχουμε:

```
   s1( "test text" );
```

οι εντολές:

```
   try {
      cout << s1.at(17) << endl;
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):

```
std::out_of_range
position beyond end of string in function: basic_string::at(size_t)
 index: 17 is greater than max_index: 9
```

Στην πραγματικότητα, τουλάχιστον στην περίπτωση αυτή, πρόκειται για ειδική περίπτωση του "`domain_error`".

---

[17] Τι είναι τα "`-14`" και "`-15`"; Δεν ξέρεις; Κεφ. 17!

24.6.2 Η Κλάση *runtime_error* και οι Παράγωγές της

Η *runtime_error* ορίζεται στο `stdexcept`:

```
class runtime_error : public exception
{
public:
   explicit runtime_error( const string& what_arg );
};
```

Όπως βλέπεις, κληρονομεί την *exception* με τον ίδιο τρόπο που την κληρονομεί και η *logic_error*.

Σε τι διαφέρει από τη *logic_error*; Η *logic_error* (ή κάποια παράγωγή της) ρίχνεται όταν ξεκινάμε έναν υπολογισμό με τιμή(-ές) κάποιας(-ων) παραμέτρου (-ων) που δεν είναι αποδεκτή(-ές). Η *runtime_error* ρίχνεται κατά τη διάρκεια του υπολογισμού, όταν κάποιο αποτέλεσμα –τελικό ή ενδιάμεσο– έχει κάποιο πρόβλημα.

24.6.2.1 Η Κλάση *range_error*

Η *range_error* είναι παράγωγη της *runtime_error*:

```
class range_error : public runtime_error
{
public:
   explicit range_error( const string& what_arg );
};
```

Ρίχνουμε εξαίρεση αυτού του τύπου όταν έχουμε τιμή συνάρτησης εκτός πεδίου τιμών. Αυτό έχει συχνά να κάνει με τη διαφορά που υπάρχει μεταξύ του πεδίου τιμών της συνάρτησης και του τύπου που παριστάνει αυτό το σύνολο στον υπολογιστή. Για παράδειγμα, αν το πεδίο τιμών μιας συνάρτησης f είναι το $\mathbb{R}$,

$f: D \to \mathbb{R}$

όταν την υλοποιήσουμε με την f_C θα το παραστήσουμε με τον `float` ή τον `double` ή τον `long double`:

f_C: $D_C \to$ `float` (ή `double` ή `long double`) (C)

Έτσι όμως κάποιες τιμές της συνάρτησης που είναι στο $\mathbb{R}$ μπορεί να μην είναι δυνατόν να παρασταθούν στον τύπο που επιλέξαμε, δηλαδή έχουμε *υπερχείλιση*. Με βάση τη (C) έχουμε πρόβλημα πεδίου τιμών.

Χαρακτηριστικό παράδειγμα είναι η συνάρτηση που *addInt()* που γράψαμε στην §17.3 για να αντιμετωπίσουμε την «ύπουλη» υπερχείλιση στις προσθέσεις ακεραίων. Το σύνολο των ακεραίων $\mathbb{Z}$ είναι κλειστό ως προς την πρόσθεση: το άθροισμα ακεραίων είναι πάντοτε ακέραιος. Αλλά το άθροισμα δύο τιμών τύπου `int` δεν είναι σίγουρο ότι μπορεί να παρασταθεί στον `int`. Εκεί θα μπορούσαμε να ρίχνουμε εξαίρεση τύπου *range_error* αντί για *IntOvrflXptn*.

24.6.2.2 Οι Κλάσεις *overflow_error* και *underflow_error*

Στην *addInt()* θα μπορούσαμε να ρίχνουμε και εξαίρεση κλάσης *overflow_error* που είναι και αυτή παράγωγη της *runtime_error*:[18]

```
class overflow_error : public runtime_error
{
public:
   explicit overflow_error( const string& what_arg );
};
```

Για τις πράξεις στους τύπους κινητής υποδιαστολής υπάρχει και το πρόβλημα της *υποχείλισης* για την οποίαν λέγαμε στην §17.9 «*Συνήθως, ο υπολογιστής σε μια τέτοια περί-*

[18] Πάντως, για ακέραιους τύπους προτιμούμε τον όρο "range error"· ο όρος *υπερχείλιση* αναφέρεται συνήθως σε τύπους κινητής υποδιαστολής.

πτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!»[19] Η κλάση

```
class underflow_error : public runtime_error
{
public:
   explicit underflow_error( const string& what_arg );
};
```

μας δίνεται για τέτοιες περιπτώσεις.

Για να δούμε ένα παράδειγμα χρήσης αυτών των κλάσεων γράφουμε μια *multDbl()* που –ακολουθώντας την *addInt()*– πολλαπλασιάζει δύο τιμές τύπου **double** αλλά έχει πρόβλεψη για υπερχείλιση και υποχείλιση:

```
double multDbl( double x, double y )
{
   double fv;

   if ( fabs(x) >= 1 )
   {
      if ( fabs(y) < 1 )  fv = x * y;
      else if ( fabs(y) <= DBL_MAX / fabs(x) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw overflow_error( ssout.str() );
      }
   }
   else    // |x| < 1
   {
      if ( fabs(y) >= 1 )  fv = x * y;
      else if ( fabs(x) >= DBL_MIN / fabs(y) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw underflow_error( ssout.str() );
      }
   }
   return fv;
} // multDbl
```

Αν δώσεις:

```
  try
  {
     cout << multDbl( DBL_MAX, DBL_MAX ) << endl;
  }
  catch( exception& x )
  {
     cout << typeid(x).name() << endl << x.what() << endl;
  }
```

θα πάρεις (g++, Dev-C++):

```
St14overflow_error
η multDbl κλήθηκε με ορίσματα 1.79769e+308, 1.79769e+308
```

Αν δώσεις:[20]

```
      cout << multDbl( DBL_MIN, DBL_MIN ) << endl;
```

---

[19] Αλλά στην §17.15 δίνουμε δύο παραδείγματα για το πώς να *αποφύγουμε* την υποχείλιση με περισσότερη προσοχή στους υπολογισμούς!

[20] Αν δώσεις απλώς "`cout << DBL_MIN*DBL_MIN << endl`" θα πάρεις "`0`".

θα πάρεις (g++, Dev-C++):

```
St15underflow_error
η multDbl κλήθηκε με ορίσματα 2.22507e-308, 2.22507e-308
```

24.6.3 Να Χρησιμοποιούμε Αυτές τις Κλάσεις;

Όπως είδες, οι κλάσεις εξαιρέσεων της C++ έχουν διαφορετική κατηγοριοποίηση από αυτές που χρησιμοποιούμε εμείς:

- Υπάρχει μια βασική κλάση, η *exception*.
- Από αυτήν παράγονται άλλες κλάσεις για τα διάφορα είδη προβλημάτων.
- Η πληροφορία για το είδος του προβλήματος υπάρχει στο όνομα της κλάσης και οποιαδήποτε άλλα σχετικά στοιχεία διαβιβάζονται μέσω της μεθόδου *what()*.

Αν σε βολεύουν μπορείς να τις χρησιμοποιείς και ακόμη μπορείς να ορίσεις δικές σου παράγωγες κλάσεις για πιο εξειδικευμένα προβλήματα.

24.7 Πώς να Σχεδιάζεις Δικές σου Κλάσεις Εξαιρέσεων

Στις σελίδες των βιβλιοθηκών Boost μπορείς να βρεις οδηγίες για τη διαχείριση λαθών και εξαιρέσεων (Abrahams 2010). Μεταξύ αυτών υπάρχουν και συμβουλές για το πώς να γράφεις δικές σου κλάσεις εξαιρέσεων. Να μια περίληψη:

- **Κάνε τις κλάσεις εξαιρέσεων παράγωγες της `std::exception`.**

  Πρόσεξε τα παραδείγματα που δώσαμε στις προηγούμενες παραγράφους: βάζουμε "`catch(exception& x)`" και με την "`typeid(x).name()`" βρίσκουμε τον τύπο της εξαίρεσης. Έτσι, αν ξέρεις ότι όλες οι κλάσεις εξαιρέσεων του προγράμματός σου είναι παράγωγες της *std::exception* μπορείς να χρησιμοποιείς την "`catch(exception& x)`" αντί για την "`catch(...)`".
- **Χρησιμοποίησε εικονική κληρονομιά** ώστε οι κλάσεις εξαιρέσεων να μπορούν να κληρονομούν δύο ή περισσότερες κλάσεις χωρίς πρόβλημα.

  Ε, τώρα, μη πάρεις και πολύ στα σοβαρά αυτή τη συμβουλή... «*Κλάση εξαιρέσεων που να κληρονομεί δύο ή περισσότερες κλάσεις εξαιρέσεων*»;! Αν τύχει να σου χρειαστεί κάτι τέτοιο μην ξεχάσεις να βάλεις το "`virtual`"!
- **Μη βάζεις στην κλάση εξαιρέσεων μέλη κλάσης `std::string` ή άλλης κλάσης με δημιουργό που μπορεί να ρίξει εξαίρεση**.

  Παρόμοια συμβουλή δώσαμε στην §24.4.2.
- **Μορφοποίησε το μήνυμα της `what()` (μόνο) αν ζητηθεί.** Γενικώς η μορφοποίηση χρειάζεται μνήμη και χρόνο και... μπορεί να φύγει και κάποια εξαίρεση.
- **Μην ασχολείσαι πολύ με το μήνυμα της `what()`**. Είναι καλό βέβαια να δώσεις στον προγραμματιστή τη δυνατότητα να καταλάβει τι συμβαίνει, αλλά είναι μάλλον απίθανο να μπορέσεις να συνθέσεις ένα μήνυμα με τέτοια «προσόντα» όταν ρίχνεις την εξαίρεση.
- **Δώσε πληροφορίες στη διεπαφή (public) της κλάσης εξαιρέσεων για την αιτία του σφάλματος**. Η προσήλωση στο μήνυμα της `what()` πιθανότατα σημαίνει ότι αμελείς την παρουσίαση (χρήσιμων) πληροφοριών.
- **Δώσε στην κλάση εξαιρέσεων «ανοσία» σε διπλή καταστροφή**. Δυστυχώς μερικοί μεταγλωττιστές, σε ορισμένες περιπτώσεις, θα βάλουν εντολές που καταστρέφουν δύο

φορές το αντικείμενο της εξαίρεσης. Αν η κλάση εξαιρέσεων έχει βέλη μην ξεχνάς, στον καταστροφέα, μετά τη "`delete`" να βάλεις και τιμή "`0`" στο κάθε βέλος.[21]

24.8 Οι Δικές μας Κλάσεις Εξαιρέσεων

Στις κλάσεις εξαιρέσεων που χρησιμοποιούμε στα μαθήματά μας έχουμε διαφορετική κατηγοριοποίηση από αυτήν που βλέπουμε στις κλάσεις της C++:

- Μια κλάση εξαιρέσεων για κάθε κλάση. Και ακόμη: αν η κλάση *D* είναι παράγωγη της κλάσης *B* τότε και η κλάση εξαιρέσεων της *D* είναι παράγωγη της κλάσης εξαιρέσεων της *B*.
- Μια κλάση εξαιρέσεων για κάθε βιβλιοθήκη.
- Μια κλάση εξαιρέσεων για κάθε πρόγραμμα.

  Ακολουθούμε τις συμβουλές της προηγούμενης παραγράφου;
- Μέχρι τώρα οι κλάσεις μας δεν είναι παράγωγες της *std::exception*. Από εδώ και πέρα θα είναι! Για παράδειγμα, οι κλάσεις εξαιρέσεων του Project 6 θα είναι:

```
struct CourseXptn :  public exception { /* . . . */ };
struct CourseCollectionXptn :  public exception { /* . . . */ };
struct StudentXptn :  public exception { /* . . . */ };
struct StudentCollectionXptn :  public exception { /* . . . */ };
struct StudentInCourseXptn :  public exception { /* . . . */ };
struct StudentInCourseCollectionXptn :  public exception { /* . . . */ };
struct MyTmpltLibXptn :  public exception { /* . . . */ };
struct ProgXptn :  public exception { /* . . . */ };
```

  Σε όλες αυτές τις κλάσεις θα πρέπει να βάλεις και τον υπερισχύοντα κατάστροφέα, π.χ. για την *CourseXptn*:

```
virtual ~CourseXptn() throw() { };
```

  Αυτό θα πρέπει να επαναληφθεί και για την *OfferedCourseXptn* που κληρονομεί την *CourseXptn*:

```
virtual ~OfferedCourseXptn() throw() { };
```

  Ακόμη, στη **main()**, αντί για την "`catch(...)`" βάζουμε:

```
catch( exception& x )
{
   cout << "unexpected exception " << typeid(x).name() << endl;
}
```

- Έχουμε επιλέξει να μην βάζουμε στις κλάσεις εξαιρέσεων
  - μέλη-βέλη προς δυναμικές μεταβλητές και
  - μέλη κλάσεων που να χρειάζονται δημιουργούς αντιγραφής.
- Το μορφοποιημένο μήνυμα προς τον χρήστη συντίθεται μετά την σύλληψη του αντικειμένου της εξαίρεσης. Με τη *what()* δεν ασχολούμαστε.
- Όταν ρίχνουμε μια εξαίρεση φροντίζουμε στο αντικείμενό της να υπάρχει
  - το όνομα της συνάρτησης από όπου ρίχτηκε η εξαίρεση και
  - οι «ένοχες» τιμές που την προκάλεσαν.
  - Ακόμη, όταν η εξαίρεση ρίχνεται από μέθοδο ενός αντικειμένου έχουμε και το κλειδί του αντικειμένου.

  Έτσι, δίνουμε αρκετές πληροφορίες για τον εντοπισμό και τη διόρθωση λαθών του προγράμματος. Δίνουμε επίσης τα δεδομένα για διορθωτικές πράξεις κατά τη διάρκεια της εκτέλεσης του προγράμματος, αν κάτι τέτοιο είναι δυνατό.

---

[21] Αυτή η συμβουλή αμφισβητείται διότι δεν φαίνεται να υπάρχουν –τώρα πια– μεταγλωττιστές που κάνουν «διπλή καταστροφή» του αντικειμένου της εξαίρεσης.

Αυτά σε σχέση με τις συμβουλές της προηγούμενης παραγράφου. Θα πρέπει όμως να βελτιώσουμε περισσότερο τις κλάσεις εξαιρέσεων. Ήδη στην §19.4 είχαμε επισημάνει:

- *«Η εξαίρεση θα μας φέρει τη μέθοδο που εμφανίστηκε το πρόβλημα, το είδος του προβλήματος και τις τιμές που το προκάλεσαν ... αλλά δεν θα μας πει σε ποιο αντικείμενο έγιναν όλα αυτά. ... Μια πρώτη σκέψη είναι να αντιγράφουμε στην εξαίρεση και το κλειδί του αντικειμένου που έχει το πρόβλημα. Αλλά πολύ συχνά αυτό δεν είναι αρκετό. ... Αν εξοπλίσουμε κάθε αντικείμενο με ένα επιπλέον μέλος, μια ταυτότητα αντικειμένου που θα την αντιγράφουμε και στο αντικείμενο της εξαίρεσης λύνουμε το πρόβλημά μας.»*
- *«Το δεύτερο πρόβλημα μπορεί να το έχεις αντιμετωπίσει ήδη αν έγραψες κάπως μεγάλα προγράμματα: "Ναι, η εξαίρεση ρίχτηκε από τη συνάρτηση myFunc, αλλά η myFunc καλείται στο πρόγραμμά μου σε 37 διαφορετικές «διαδρομές» εκτέλεσης!"»*

Ήδη, στο Project 6, είδαμε αντικείμενα εξαιρέσεων με το κλειδί του αντικειμενου· πάντως, όπως υποσχεθήκαμε και στην §19.4 *δεν θα δώσουμε παράδειγμα με ταυτότητα αντικειμένου.*

24.8.1 Δύο Μέθοδοι για τις Κλάσεις Εξαιρέσεων

Θα ασχοληθούμε όμως με το δεύτερο πρόβλημα ξεκινώντας από ένα παράδειγμα της §21.6.1. Εκεί, γράφοντας την επιφόρτωση του "`operator=`"της *BString* με βάση τον δημιουργό αντιγραφής, είχαμε γράψει:

```
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
```

Δηλαδή: πιάναμε την όποια εξαίρεση *BStringXptn*, διορθώναμε το *funcName* και την ξαναρίχναμε. Έτσι, φαίνεται ότι την εξαίρεση τη ρίχνει ο "`operator=`".

Το σωστό όμως θα ήταν να πούμε ότι *την εξαίρεση τη ρίχνει ο δημιουργός όταν καλείται από τον* "`operator=`". Θα μπορούσαμε να γράψουμε:

```
      catch( BStringXptn& x )
      {  strcat( x.funcName, "|operator=" );
         throw;  }
```

Έτσι, η *x.funcName* έχει τιμή `"BString|operator="`. Και αν εξοπλίσουμε τη *BStringXptn* με μια:

```
   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                                 else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
```

από τις

```
// . . .
         case BStringXptn::noMemory:
            cout << "out of memory "; x.displayFuncName( cout );
            break;
// . . .
```

θα μπορούσαμε να πάρουμε:

```
out of memory thrown in BString
 called by operator=
```

Καλό θα είναι να εξοπλίσουμε τη *BStringXptn* και με μια

```
   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
```

```
      strcat( funcName, fn );  } // appendFuncName
```

και έτσι θα γράφουμε:

```
BString& BString::operator=( const BString& rhs )
{
   if ( &rhs != this )
   {
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  x.appendFuncName( "operator=" );
         throw;  }
   }
   return *this;
} // BString::operator=
```

Η *BStringXptn* τώρα έχει γίνει:

```
struct BStringXptn :  public exception
{
   static const int fnLength = 100;
   enum { noMemory, outOfRange };
   char funcName[ fnLength ];
   int  errorCode;
   int  errorValue;
   BStringXptn( const char* fn, int ec, int ev = 0 )
   {  strncpy( funcName, fn, fnLength-1 );
                                    funcName[fnLength-1] = '\0';
      errorCode = ec;  errorValue = ev;  }
   virtual ~BStringXptn() { };

   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
      strcat( funcName, fn );  } // appendFuncName

   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                              else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
}; // BStringXptn
```

Εκτός από τις δύο μεθόδους πρόσεξε και κάτι άλλο: Αντικαταστήσαμε τη «μαγική σταθερά» "`100`" με "`fnLength`" που ορίζεται ως:

```
   static const int fnLength = 100;
```

Αν σκεφτείς ότι μπορεί να χρειαστεί να αποθηκεύεις στη *funcName* όλα τα βήματα-κλήσεις συναρτήσεων και στα ονόματα συναρτήσεων-μελών να βάζεις και το όνομα της κλάσης –π.χ. όχι "`appendFuncName("load")`" αλλά πιο συγκεριμένα "`appendFuncName( "Course::load")`"– το "`100`" μπορεί να μην είναι αρκετό.

Ένα δεύτερο παράδειγμα χρήσης αυτών των εργαλείων μπορούμε να δούμε στον δημιουργό της *DateTime* όπως τον ξαναγράψαμε στην §24.5.

Αλλάζουμε την κλάση εξαιρέσεων:

```
struct DateXptn :  public exception
{
   static const int fnLength = 100;
   enum { yearErr, monthErr, dayErr, outOfLimits,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[ fnLength ];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
```

```
   Date errDateVal;
   DateXptn( const char* fn, int ec,
             int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : errorCode(ec), errVal1(ev1), errVal2(ev2), errVal3(ev3)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   DateXptn( const char* fn, int ec, int ev1, const Date& ed )
      : errorCode(ec), errVal1(ev1), errDateVal(ed)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   virtual ~DateXptn() throw() {  };

   void appendFuncName( const char* fn )
   // ΟΠΩΣ ΣΤΗ BStringXptn

   void displayFuncName( ostream& tout )
   // ΟΠΩΣ ΣΤΗ BStringXptn
}; // DateXptn
```

και τον δημιουργό της *DateTime*:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}
catch( DateTimeXptn& x )
{  throw;  }
catch( DateXptn& x )
{
   x.appendFuncName( "DateTime" );
   throw;
} // DateTime::DateTime
```

Τώρα μπορεί να έχουμε ένα μήνυμα σαν

```
day error ( 44 ) thrown in Date
 called by DateTime
```

που είναι ακριβέστερο από το να πούμε απλώς ότι «*βρήκαμε λάθος στον δημιουργό της DateTime.*»

Ας δούμε τώρα πώς θα αξιοποιήσουμε αυτά τα εργαλεία. Μήπως θα πρέπει να ξαναγράψουμε όλες τις συναρτήσεις και όλες τις μεθόδους ώστε να πιάνουμε ολόκληρη την ακολουθία κλήσεων; Ούτε να το σκέφτεσαι!

♦ ***Η διαδικασία «πιάσε – διόρθωσε – ξαναρίξε» είναι χρονοβόρα.***

Δεν μπορείς επομένως να την έχεις παντού σε ένα πρόγραμμα που είναι σε εκμετάλλευση. Και αν μεν η ρίψη της εξαίρεσης θα έχει αποτέλεσμα την διακοπή της εκτέλεσης του προγράμματος οι καθυστερήσεις δεν μας πειράζουν. Αν όμως υπάρχει δυνατότητα να κάνεις διόρθωση και να συνεχίσεις την εκτέλεση το πράγμα αλλάζει.

Όταν κάνεις την ανάπτυξη του προγράμματος και ψάχνεις για να εντοπίσεις και να διορθώσεις μερικά –ενδεχομένως δύσκολα– λάθη αυτά τα εργαλεία μπορεί να σε βοηθήσουν.

Μπορεί να τα χρησιμοποιήσεις και στο πρόγραμμα που είναι σε εκμετάλλευση αλλά σε σημεία που θα επιλέξεις με προσοχή.

Σε κάθε περίπτωση πάρε υπόψη σου και το εξής: *Υπάρχουν ακολουθίες κλήσεων που ταυτοποιούνται με μοναδικό τρόπο από την αρχή και το τέλος.*

24.9 Ασφάλεια ως προς τις Εξαιρέσεις

Στις §21.6 και §21.6.1 είδαμε δύο μορφές της `BString::operator=()` που είπαμε ότι έχουν **ασφάλεια προς τις εξαιρέσεις** (exception safety) επιπέδου **ισχυρής εγγύησης** (strong guarantee). Όπως είπαμε, με τον όρο αυτόν εννοούμε ότι αυτή η μέθοδος (επιφόρτωσης) έχει τα εξής χαρακτηριστικά: σε περίπτωση που θα εγερθεί εξαίρεση όταν εκτελείται η "`a = b`"

- Δεν θα έχουμε διαρροή μνήμης.
- Δεν θα αλλάξει η τιμή της *a*.

Η ασφάλεια ισχυρής εγγύησης της δεύτερης μορφής στηρίζεται στη σιγουριά μας ότι η *BString::swap()* δεν θα ρίξει εξαίρεση. Λέμε ότι η *BString::swap()* έχει **ασφάλεια προς τις εξαιρέσεις** σε επίπεδο **εγγύησης μη-ρίψης** (no-throw guarantee).

Στην §20.7.2.3 είδαμε την

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

που «σβήνει» το στοιχείο που βρίσκεται στη θέση *ndx* σε έναν ταξινομημένο πίνακα. Λέγαμε όμως ότι κάποια από τις "`rAllStops[k] = rAllStops[k+1]`" της *erase1RouteStop()* μπορεί να ρίξει *bad_alloc* διότι τα `rAllStops[k]`, που είναι κλάσης *RouteStop*, έχουν μέλος "`string sName`".

Ας πούμε λοιπόν ότι ρίχνεται η εξαίρεση όταν ο δείκτης *k* έχει τιμή k_0. Ο τελεστής εκχώρησης της *string* έχει ασφάλεια ισχυρής εγγύησης που σημαίνει:

- δεν έχουμε διαρροή μνήμης και
- το `rAllStops[`k_0`]` κρατάει την αρχική του τιμή. Επομένως:
  - Αν k_0 `==` *ndx* σημαίνει ότι ο πίνακας έχει μείνει αθικτος και η αναλλοίωτη της κλάσης ισχύει.
  - Αν k_0 > *ndx*, στο προηγούμενο βήμα της **for** το `rAllStops[`k_0`-1]` έχει πάρει την τιμή του `rAllStops[`k_0`]`. Έτσι, ενώ έχουμε σβήσει το `rAllStops[`*ndx*`]`, το πλήθος των στοιχείων μεταξύ των δύο φρουρών παραμένει αυτό που ήταν αφού τα στοιχεία στις θέσεις k_0-1 και k_0 έχουν την ίδια τιμή. Αυτή η κατάσταση παραβιάζει την αναλλοίωτη της κλάσης και συγκεκριμένα το κομμάτι:

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

Πάντως: και στις δύο περιπτώσεις το βέλος `rAllStops` δείχνει τον δυναμικό πίνακα, το *rReserved* μας λέει πόση δυναμική μνήμη έχουμε πάρει και το *rNoOfStops* πόση από αυτήν είναι σε χρήση. Δηλαδή το αντικείμενο είναι διαχειρίσιμο είτε με τον καταστροφέα είτε με την *Route::clearRouteStops()*.

Λέμε ότι η *Route::erase1RouteStop()* έχει **ασφάλεια** επιπέδου **βασικής εγγύησης** (basic guarantee).[22]

Να τα τρία επίπεδα εγγύησης ασφάλειας μιας μεθόδου (ή μιας συνάρτησης που διαχειρίζεται ένα αντικείμενο) ως προς τις εξαιρέσεις (Abrahams 2001):

- Η *βασική* εγγύηση: *Δεν υπάρχει διαρροή πόρων και η αναλλοίωτη του αντικειμένου διατηρείται*. Το ότι ισχύει η αναλλοίωτη σημαίνει ότι μπορεί να κληθεί ο καταστροφέας (ας πούμε με το ξετύλιγμα της στοίβας) που έχει την αναλλοίωτη ως προϋπόθεση. Κατά τα άλλα, αυτό δεν λέει και πολλά πράγματα διότι καταστάσεις που να ισχύει η αναλλοίωτη μπορεί να υπάρχουν πολλές. Αν θέλεις να συνεχίσεις να δουλεύεις με το αντικείμε-

[22] Μερικοί ονομάζουν αυτό το επίπεδο **ελάχιστης** (minimal) εγγύησης διότι δεν ισχύει ολόκληρη η αναλλοίωτη.

νο θα πρέπει να το φέρεις σε συγκεκριμένη κατάσταση π.χ. να κάνεις κάποιο είδος επανεκκίνησης.

- Η *ισχυρή* εγγύηση: Η εκτέλεση της μεθόδου
  - είτε *ολοκληρώνεται επιτυχώς*
  - είτε *διακόπτεται με έγερση εξαίρεσης αλλά το πρόγραμμα* (και το αντικείμενο) *βρίσκεται στην κατάσταση που ήταν πριν αρχίσει η εκτέλεση της μεθόδου. Δεν υπάρχει διαρροή πόρων.*
- Η εγγύηση *μη-ρίψης*: *Η εκτέλεση της μεθόδου ολοκληρώνεται επιτυχώς χωρίς να ριχτεί (ή να περάσει μέσα από αυτήν) εξαίρεση.*

Προφανώς για κάθε μέθοδο θα θέλαμε να έχουμε το μέγιστο δυνατό επίπεδο ασφάλειας. Ποιο ακριβώς είναι το επίπεδο εξαρτάται από τη συγκεκριμένη μέθοδο. Και πώς πετυχαίνουμε το επίπεδο που θέλουμε; Ας ξαναγυρίσουμε, για παράδειγμα, στη *Route*::*erase1-RouteStop* και ας πούμε ότι προσπαθούμε να την κάνουμε να έχει ισχυρή εγγύηση ασφάλειας. Μπορούμε να ξαναδώσουμε την αρχική τιμή στα στοιχεία που αλλάξαμε; Ούτε λόγος! Θα πρέπει να κάνουμε αντιγραφές που –λογικώς– θα ρίξουν εξαίρεση. Ο μόνος τρόπος είναι ο εξής: Να μην πειράξουμε τον αρχικό πίνακα αλλά να τον αντιγράψουμε σε έναν άλλον

```
RouteStop* tmp( new RouteStop[rReserved] );
```

παραλείποντας το `rAllStops[`*ndx*`]`:

- Αν η αντιγραφή ολοκληρωθεί επιτυχώς ανταλλάσσουμε τις τιμές των βελών *rAllStops* και *tmp*.
- Αλλιώς, αν ριχτεί εξαίρεση, ο πίνακας παραμένει όπως ήταν.

Δηλαδή: για να διαγράψουμε μια τιμή αντιγράφουμε ολόκληρον τον πίνακα. Είναι δεκτό αυτό; Αλλά και τώρα μήπως κάνουμε κάτι καλύτερο; Κατά μέσον όρο αντιγράφουμε τον μισό πίνακα! Γενικώς η αντιγραφή πίνακα –για διαγραφή ή εισαγωγή μιας τιμής– φαίνεται απαράδεκτη αλλά δεν έχουμε και άλλη επιλογή (πάντως εδώ οι πίνακες δεν είναι και τόσο μεγάλοι).

Δηλαδή το καλό επίπεδο ασφάλειας για τις εξαιρέσεις πληρώνεται με (μεγάλο) υπολογιστικό χρόνο; Όχι απαραιτήτως. Η λύση στο πρόβλημά μας βρίσκεται στην αλλαγή δομής δεδομένων για τις στάσεις που αποτελούν ένα σύνολο: όπως θα δεις αργότερα, χρησιμοποιώντας τα εργαλεία που μας δίνει η STL, θα τις βάλουμε σε ένα σύνολο με στοιχεία τύπου *RouteStop* ή, όπως θα το γράφουμε, "`set<RouteStop>`"· σε αυτή τη δομή οι εισαγωγές και οι διαγραφές στοιχείων γίνονται με ισχυρή εγγύηση ασφάλειας χωρίς να είναι χρονοβόρες.

Επομένως

♦ ***Για να έχουμε το επιθυμητό επίπεδο εγγύησης ασφάλειας πρέπει να σχεδιάσουμε καταλλήλως το πρόγραμμά μας.***

Για να διασφαλίσουμε τη μη-διαρροή πόρων χρησιμοποιούμε τις **try**/**catch**. Η τεχνική RAII είναι σαφώς καλύτερη και ασφαλέστερη. Με τα εργαλεία της STL, που θα μάθουμε αργότερα, η εφαρμογή της θα γίνει απλούστερη.

Ένα εργαλείο, πολύ συχνά χρήσιμο, είναι η (ασφαλής) *swap()*. Μερικές φορές θα πρέπει να συνδυάσεις τη χρήση της με την τεχνική PIMPL (§22.10).

24.10 Σύνοψη

Η C μας δίνει τη δυνατότητα να βρούμε τί δεν πήγε καλά όταν καλέσαμε μια από τις συναρτήσεις της βιβλιοθήκης της

- από την τιμή που επιστρέφει η συνάρτηση και

- την τιμή της καθολικής μεταβλητής *errno*.

Μας δίνει τη δυνατότητα να διακόψουμε την εκτέλεση του προγράμματος (*abort()*, *exit()*, *_Exit()*) και να καθορίσουμε τι θα συμβεί στην περίπτωση αυτή (*atexit()*).

Με την *assert()* μπορούμε να ελέγχουμε τις συνθήκες επαλήθευσης στα διάφορα σημεία εκτέλεσης του προγράμματος. Αλλά, όπως ξέρουμε, αν η συνθήκη δεν ισχύει η «ποινή είναι βαριά».

Για τις εξαιρέσεις της C++ μάθαμε ότι

- Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται η *terminate()* (που καλεί την *abort()*) για να διακόψει την εκτέλεση του προγράμματος.
- Αν παραβιασθεί μια προδιαγραφή εξαιρέσεων –δηλαδή αν από μια συνάρτηση ριχτεί εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση *unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Οι *set_terminate()* και *set_unexpected()* σου επιτρέπουν να αντικαταστήσεις τις *terminate()* και *unexpected()* αντιστοίχως.

Αν έλθουμε στο «πρακτέο» θα πρέπει να πούμε κατ' αρχάς ότι καλό είναι να αφήσεις κατά μέρος τις προδιαγραφές εξαιρέσεων εκτός από την "`throw()`".

Αν γράφεις δικές σου κλάσεις εξαιρέσεων (ή χρησιμοποιείς τις δικές μας) καλό είναι να τις βάζεις να κληρονομούν (αμέσως ή εμμέσως) την *exception*.

Όταν γράφεις το πρόγραμμά σου στόχος σου θα πρέπει να είναι η καλύτερη δυνατή εγγύηση ασφάλειας ως προς τις εξαιρέσεις. Η τεχνική RAII και η ασφαλής *swap()* είναι εργαλεία που μπορείς να χρησιμοποιήσεις.

Πάντως

- η ασφάλεια ως προς τις εξαιρέσεις και
- το πού θα πιάνεις και πώς θα χειρίζεσαι τις εξαιρέσεις

έχουν σχέση (και) με τη σχεδίαση του προγράμματός σου.

κεφάλαιο

25

Περιγράμματα Κλάσεων

Ο στόχος μας σε αυτό το κεφάλαιο:

Θα μάθουμε πώς μπορούμε να υλοποιήσουμε δομές δεδομένων με προγραμματιστικά αντικείμενα της C++: τα περιγράμματα κλάσεων.

Προσδοκώμενα αποτελέσματα:

Όπως με ένα περίγραμμα συνάρτησης μπορείς να υλοποιήσεις προγραμματιστικώς έναν αλγόριθμο, με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων ώστε να μπορείς να έχεις στιγμιότυπα για τους τύπους που σε ενδιαφέρουν.

Έννοιες κλειδιά:

- *περίγραμμα κλάσης*
- *παράμετρος περιγράμματος*
- *εξειδίκευση περιγράμματος*
- *μερική εξειδίκευση περιγράμματος*
- *περιγράμματα και κληρονομιές*
- *περιέχουσα κλάση*
- *περιεχόμενη κλάση*
- *έξυπνα βέλη*
- *κλάσεις χαρακτηριστικών*

Περιεχόμενα:

25.1 Από την Κλάση *BString* στο Περίγραμμα ... 952
25.2 Φίλες Συναρτήσεις Περιγραμμάτων ... 955
25.3 Καθολικές Συναρτήσεις για Περιγράμματα ... 956
25.3.1 Ένα Απλό Παράδειγμα: *pair* ... 956
25.4 Κατανομή σε Αρχεία ... 958
25.5 Παράμετροι και Εξειδικεύσεις ... 959
25.5.1 Μερική Εξειδίκευση Περιγράμματος Κλάσης ... 961
25.6 Περιγράμματα και Κληρονομιές ... 964
25.6.1 Κληρονομιά Κλάσης από Περίγραμμα Κλάσης ... 964
25.6.2 Κληρονομιά Περιγράμματος Κλάσης από Κλάση ... 964
25.6.3 Κληρονομιά Περιγράμματος Κλάσης από Περίγραμμα ... 965
25.7 Περιέχουσες Κλάσεις ... 966
25.7.1 Το Περίγραμμα μιας Λίστας ... 970
25.7.1.1 Ο Καταστροφέας ... 974
25.7.2 Η Περιεχόμενη Κλάση ... 975
25.7.3 Ένα Πρόβλημα, μια Λύση και ένα Άλλο Πρόβλημα ... 975
25.7.4 Μια Άλλη Στοίβα ... 976
25.8 Έξυπνα Βέλη ... 977
25.8.1 *std::auto_ptr* ... 983
25.8.2 Συνελόντι Ειπείν... ... 985
25.9 Ένα Χρήσιμο Περίγραμμα Κλάσης ... 985

25.10 Ανακεφαλαίωση 988
Ασκήσεις 988

Εισαγωγικές Παρατηρήσεις:
Όπως είναι χρήσιμο να έχουμε *περιγράμματα συναρτήσεων*, όπου μεταφράζουμε σε C++ γενικούς αλγόριθμους με τους τύπους των στοιχείων να είναι παράμετροι, είναι χρήσιμο να έχουμε και **περιγράμματα κλάσεων**[1] (class templates) στα οποία μπορούμε να μεταφράσουμε σε C++ δομές δεδομένων και τους αλγόριθμους για τον χειρισμό τους.

Παρόμοιες δυνατότητες προσφέρουν και άλλες γλώσσες προγραμματισμού και – μεταξύ αυτών– τα άλλα «παιδιά της C»:

- Η Java έχει τους **γενικούς τύπους** (generic types).
- Η C# έχει τις **γενικές κλάσεις** (generic classes).

Ο τύπος (*std*::)*string*, που χρησιμοποιούμε συνεχώς στα προγράμματά μας, είναι ένα στιγμιότυπο του περιγράμματος *basic_string*. Ακολουθώντας και εμείς αυτήν την ιδέα(!) θα ξεκινήσουμε την παρουσίαση των περιγραμμάτων κλάσεων ως εξής:

- Από τη *BString* θα πάρουμε το περίγραμμα κλάσης *BStringT*.
- Θα ξαναπάρουμε την αρχική κλάση ως στιγμιότυπο του περιγράμματος.

25.1 Από την Κλάση *BString* στο Περίγραμμα

Στα προηγούμενα κεφάλαια αναπτύξαμε το παράδειγμα της κλάσης *BString*. Επειδή, η κλάση αυτή είναι χρήσιμη γενικότερα, θα θέλαμε να την μετατρέψουμε σε *περίγραμμα* ώστε να μπορούμε να τη χρησιμοποιήσουμε, όχι μόνον με τον `char`, αλλά και με άλλους τύπους, όπως ο `unsigned char` ή ο `wchar_t`. Η C++ μας δίνει τη δυνατότητα να έχουμε, εκτός από περιγράμματα συναρτήσεων, και περιγράμματα κλάσεων.

Στην κλάση, όπως την έχουμε μέχρι τώρα, διαχωρίζουμε τον ερήμην δημιουργό από τον δημιουργό με αρχική τιμή:

```
 0: class BString
 1: { // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
 2: friend bool operator==( const BString& lhs,
 3:                         const BString& rhs );
 4: public:
 5:    BString();
 6:    BString( const char* rhs, int n=0 );
 7:    BString( const BString& rhs );
 8:    ~BString() {  delete[] bsData;  };
 9:    BString& operator=( const BString& rhs );
10:    BString& assign( const BString& rhs )
11:                                  {  return (*this = rhs);  }
12:    const char* c_str() const
13:                     {  bsData[bsLen] = '\0';  return bsData;  }
14:    size_type length() const {  return bsLen;  }
15:    bool empty() const {  return ( bsLen == 0 );  }
16:    char& at( int k ) const;
17:    char& operator[]( int pos ) const {  return bsData[pos];  }
18:    BString& operator+=( const BString& rhs );
19:    BString& append( const BString& rhs )
20:                                 {  return (*this += rhs);  }
21:    void swap( BString& rhs );
22:    int compare( const BString& rhs ) const;
23: private:
24:    static const unsigned int bsIncr = 16;
25:    char*     bsData;
26:    size_type bsLen;
27:    size_type bsReserved;
```

[1] Θα δεις και τον όρο **παραμετρικές** (parametric) κλάσεις.

```
 28:
 29:    static size_type cStrLen( const char* cs );
 30:    static int stringCmpr( const char* lhs, const char* rhs,
 31:                           int n );
 32: }; // BString
```

Ο ορισμός του ερήμην δημιουργού είναι:

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

Βάλαμε μέσα στη δήλωση της κλάσης τους ορισμούς των τριών μεθόδων **inline** και του καταστροφέα.

Πέρα από τη φίλη καθολική **operator==()** να θυμηθούμε ότι έχουμε και την καθολική:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

Για να πάρουμε περίγραμμα κλάσης, ας το πούμε *BStringT*, από την ειδική περίπτωση ορμαθών με στοιχεία τύπου **char**, κάνουμε τα εξής:

- Αλλάζουμε το **'\0'** σε "**char(0)**" (στη γρ. 13)
- Αντικαθιστούμε τον τύπο "**char**"[2], όπου τον βρούμε (γρ. 6, 12, 16, 17, 25, 29, 30), με μια παράμετρο, ας την πούμε "**K**".
- Αντικαθιστούμε το "**BString**", όπου τον βρούμε, με "**BStringT**".

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT& lhs,
                        const BStringT& rhs );
public:
   BStringT();
   BStringT( const K* rhs, int n=0 );
   BStringT( const BStringT& rhs );
   ~BStringT() {  delete[] bsData;  };
   BStringT& operator=( const BStringT& rhs );
   BStringT& assign( const BStringT& rhs )
                                        {  return (*this = rhs);  }
   const K* c_str() const
                          {  bsData[bsLen] = K(0);  return bsData;  }
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   K& at( int k ) const;
   K& operator[]( int pos ) const {  return bsData[pos];  }
   BStringT& operator+=( const BStringT& rhs );
   BStringT& append( const BStringT& rhs )
                                       {  return (*this += rhs);  }
   void swap( BStringT& rhs );
   int compare( const BStringT& rhs ) const;
private:
   static const unsigned int bsIncr = 16;
   K*        bsData;
   size_type bsLen;
   size_type bsReserved;
```

[2] Με το "replace" του κειμενογράφου!

```
   static size_type cStrLen( const K* cs );
   static int stringCmpr( const K* lhs, const K* rhs, int n );
}; // BStringT
```

Βλέπεις φυσικά και το –γνωστό μας από τα περιγράμματα συναρτήσεων– «καπέλο» "`template < typename K >`".

Τώρα, αντί για τη δήλωση:

```
   BString s1( "abc" ), s2, s3;
   BString bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

μπορούμε να γράψουμε:

```
   BStringT<char>  s1( "abc" ), s2, s3;
   BStringT<char>  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Λέμε δηλαδή ότι ο τύπος (κλάση) είναι ένα *στιγμιότυπο* του περιγράμματος κλάσης *BStringT* για παράμετρο-τύπο `char`.

Στη βιβλιοθήκη της C++ ορίζεται κατ' αρχάς το *basic_string* που είναι περίγραμμα κλάσης. Η *string* ορίζεται ως στιγμιότυπο του περιγράμματος:

```
typedef basic_string<char> string;
```

Παρομοίως «βαφτίζεται» και ένα άλλο στιγμιότυπο:

```
typedef basic_string<wchar_t> wstring;
```

Μιμούμενοι τα παραπάνω, μετά τον ορισμό του περιγράμματος, θα μπορούσαμε να ορίσουμε:

```
typedef BStringT< char >                 String;
```

και να γράψουμε τις παραπάνω δηλώσεις:

```
   String  s1( "abc" ), s2, s3;
   String  bs1( "abcdefg" ), bs2( "abcdefg", 3 );
```

Η *String* δεν είναι άλλη από τη *BString*.

Μπορούμε ακόμη να ορίσουμε:

```
typedef BStringT< wchar_t >             WString;
typedef BStringT< unsigned char >       UString;
```

και να δηλώσουμε:

```
   WString s1( L"abcd" ), s2( L"abcd" );
```

Να δούμε τώρα πώς γράφονται οι ορισμοί των μεθόδων. Για τον καταστροφέα, τη *c_str()*, τη *length()*, την *empty()*, την *assign()* και την *append()* δεν υπάρχει πρόβλημα· ορίζονται με τις δηλώσεις τους. Ας δούμε την `operator+=()` που επιφορτώνει τον "`+=`" και την ορίζουμε έξω από τη δήλωση της κλάσης. Όπως καταλαβαίνεις, αφού η *BStringT* έγινε περίγραμμα κλάσης και η `operator+=()` είναι πια *περίγραμμα συνάρτησης*. Φυσικά θα έχει τις ίδιες παραμέτρους με την κλάση. Άρα ο ορισμός της θα αρχίζει με τα: "`template < typename K >`".

Αν γράφαμε τη μέθοδό μας για την κλάση `BStringT<char>` η επικεφαλίδα της θα ήταν:

```
BStringT<char>& BStringT<char>::operator+=( const BStringT<char>& rhs )
```

ενώ, αν τη γράφαμε για τη `BStringT<wchar_t>` θα ήταν:

```
BStringT<wchar_t>&
                 BStringT<wchar_t>::operator+=( const BStringT<wchar_t>& rhs )
```

Μπορούμε λοιπόν να μαντέψουμε ότι όταν έχουμε στοιχεία κλάσης *K* θα πρέπει να έχουμε:

```
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
```

Πράγματι, το περίγραμμα της μεθόδου (συνάρτησης) θα είναι:

```
template < typename K >
BStringT<K>& BStringT<K>::operator+=( const BStringT<K>& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
```

```
      K* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new K[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringTXptn( "operator+=",
                             BStringTXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BStringT<K>::operator+=
```

Πρόσεξε ότι οι αλλαγές είναι

- στην επικεφαλίδα,
- στη δήλωση "`char* tmp`" που έγινε "`K* tmp`" και
- στη "`new char[tmpRes]`" που έγινε "`new K[tmpRes]`".

Να δούμε πώς γίνεται ο ερήμην δημιουργός, που είδαμε και πιο πάνω:

```
template < typename K >
BStringT<K>::BStringT()
{
   try {  bsData = new K[bsIncr];  }
   catch( bad_alloc )
   {  throw BStringTXptn( "BStringT",
                          BStringTXptn::allocFailed );  }
   bsReserved = bsIncr;
   bsLen = 0;
} // BStringT<K>::BStringT<K>
```

Πρόσεξε ότι το όνομα του δημιουργού είναι `BStringT` και όχι `BStringT<K>`.

25.2 Φίλες Συναρτήσεις Περιγραμμάτων

Ας δούμε τώρα τι γίνεται με τις φίλες συναρτήσεις: Κατ' αρχήν, δηλώνονται όπως ακριβώς ξέρουμε:

```
template < typename K >
class BStringT
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Φυσικά, και αυτές γίνονται τώρα περιγράμματα συναρτήσεων και έτσι δίνονται οι ορισμοί τους, π.χ.:

```
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int int fv( BStringT<K>::stringCmpr(lhs.bsData, rhs.bsData,
                                       min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==( const BStringT<K>&
```

Ωραία όλα αυτά, αλλά –όπως είπαμε– «κατ' αρχήν». Το πιο πιθανό είναι ότι ο μεταγλωττιστής σου θα τα απορρίψει! Γιατί; Διότι δεν έχει την «υπομονή» να περιμένει να βρεί τον ορισμό της συνάρτησης `operator==()` και μόλις βρει τη δήλωση `friend` μας λέει ότι δεν ξέρει τέτοια συνάρτηση.

Η πιο απλή λύση στο πρόβλημά μας είναι η εξής: βάζουμε τον ορισμό της συνάρτησης μαζί με τη δήλωση **friend**:

```
template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs )
{
   int fv( stringCmpr(lhs.bsData, rhs.bsData,
                      min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
public:
   BStringT();     // default constructor
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Αυτό θα περάσει από τον μεταγλωττιστή. Πάντως ορισμένοι μεταγλωττιστές (π.χ.: gcc) δεν θα καταλάβουν ότι αυτό είναι περίγραμμα συνάρτησης και θα πρέπει να δηλώσεις:

```
template < typename K1 >
friend bool operator==( const BStringT<K1>& lhs, const BStringT<K1>& rhs )
// . . .
```

Σημείωση:▶

Αν θέλεις να βάλεις τον ορισμό της φίλης συνάρτησης έξω από τον ορισμό της κλάσης θα πρέπει θα *πρέπει να βάλεις πριν από την κλάση προειδοποιητικές δηλώσεις (§22.4) της κλάσης και της συνάρτησης*:

```
template < typename K > class BStringT;
template < typename K >
bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );

template < typename K >
class BStringT
{// I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
friend bool operator==( const BStringT<K>& lhs, const BStringT<K>& rhs );
// . . .
```

Στη συνέχεια, μπορείς να βάλεις τον ορισμό της συνάρτησης όπου θέλεις.◀

25.3 Καθολικές Συναρτήσεις για Περιγράμματα

Περιγράμματα συναρτήσεων θα γίνουν και οι καθολικές συναρτήσεις που έχουν σχέση με κάποιο περίγραμμα κλάσης. Δες πώς θα γίνει η (καθολική συνάρτηση) **operator<<()** που επιφορτώνει τον "<<":

```
template < typename K >
ostream& operator<<( ostream& tout, const BStringT<K>& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BStringT<K>
```

Παρατήρηση: ▶

Θα πρέπει να παρατηρήσουμε εδώ ότι αν πάρουμε στιγμιότυπο του περιγράμματος για **wchar_t**, η **operator<<()** δεν δουλεύει όπως θέλουμε! Θα χρειαστείς εξειδίκευση της συνάρτησης (όχι της κλάσης).◀

25.3.1 Ένα Απλό Παράδειγμα: *pair*

Αν βάλεις στο πρόγραμμά σου "**#include <utility>**" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*::*pair* που βάζει σε μια **struct** δύο μεταβλητές και χρησιμοποιείται στην STL. Εδώ θα γράψουμε ένα περίγραμμα κλάσης

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;
// . . .
}; // PairT
```

που θα έχει ακριβώς τις ιδιότητες του *pair*: Μια **struct** με δύο μέλη και λεξικογραφική διάταξη. Τι θα πει αυτό; Ορίζεται ο "<" ως εξής: Αν

```
PairT<T1, T2> x, y;
```

τότε η $x < y$ ισχύει αν και μόνον αν:

$$(x.first < y.first) \;||\; (x.first == y.first \;\&\&\; x.second < y.second)$$

ή, ισοδυνάμως:

$$x.first < y.first \;||\; (!(y.first < x.first) \;\&\&\; x.second < y.second$$

Αυτή η μορφή[3] *δεν έχει σύγκριση για ισότητα* και επομένως είναι προτιμότερη αν ο *T1* είναι τύπος κινητής υποδιαστολής.

Ο ερήμην δημιουργός θα είναι:

```
PairT()
   :  first( T1() ), second( T2() ) { };
```

πράγμα που *προϋποθέτει την ύπαρξη ερήμην δημιουργών για τους T1 και T2*.

Ένας δημιουργός με αρχικές τιμές θα είναι:

```
PairT( const T1& x, const T2& y )
   :  first( x ), second( y ) { };
```

Για να μπορούμε να γράψουμε τα "**first(x)**" και "**second(y)**" θα πρέπει *οι T1 και T2 να έχουν δημιουργούς με αρχική τιμή*.

Για τύπους *U* και *V* που έχουν δημιουργούς μετατροπής του *U* στον *T1* και του *V* στον *T2* μπορούμε να γράψουμε

```
PairT( const PairT<U,V>& a )
   : first( a.first ), second( a.second ) { };
```

Και αυτό πώς μπαίνει μέσα στη δήλωση του περιγράμματος; Ως περίγραμμα μέσα σε περίγραμμα!

```
template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT
```

Όπως καταλαβαίνεις, κάθε στιγμιότυπο του *PairT* θα είναι μια κλάση που θα περιέχει ένα περίγραμμα δημιουργού.

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε τον "<" ως:

```
template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
                           (!(y.first < x.first) && x.second < y.second) );  }
```

Με περίγραμμα καθολικής συνάρτησης επιφορτώνουμε και τον "==":

[3] που την προτιμάει το πρότυπο…

```
template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }
```

όπου φυσικά δεν μπορούμε να αποφύγουμε τη σύγκριση για ισότητα όποιοι και αν είναι οι *T1*, *T2*.

Τέλος, το περίγραμμα καθολικής συνάρτησης:

```
template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }
```

ζευγαρώνει δύο τιμές *x*, *y* τύπων *T1*, *T2* αντιστοίχως.

Ας καταγράψουμε τις απαιτήσεις από τους *T1*, *T2* για να μπορούμε να ζευγαρώνουμε τιμές τους με το *PairT*: Θα πρέπει

- Να έχουν ερήμην δημιουργούς.
- Να έχουν δημιουργούς αρχικής τιμής.
- Να έχουν τον τελεστή "`<`".
- Να έχουν τον τελεστή "`==`".

Και κάτι ακόμη: χωρίς να μπορούμε να επιφορτώσουμε τον "`=`" –αφού δεν ξέρουμε τους *T1*, *T2*– θα πρέπει η εκχώρηση να γίνεται σωστά. Αυτό σημαίνει ότι: *ο τελεστής εκχώρησης "`=`" δουλεύει σωστά για τους T1, T2*.

Όσο για τη χρήση ενός τέτοιου περιγράμματος να πούμε τα εξής: Όπως θα καταλάβεις αργότερα, χρησιμοποιώντας τα εργαλεία της STL στο παράδειγμα με τους φοιτητές και τα μαθήματα, θα δημιουργηθούν ζεύγη τύπων όπως:

```
PairT< CourseKey, Course >
PairT< unsigned int, Student >
```

Βέβαια, μπορεί να σου κατέβουν και άλλες ιδέες. Θα μπορούσες να γράψεις τη *minmax* που είδαμε στην §13.9.3 (Παρ. 1) έτσι:

```
PairT< int, int > minmax( int x, int y )
{
   PairT< int, int > fv;

   if ( x < y )
      fv = PairT< int, int >( x, y );
   else // x >= y
      fv = PairT< int, int >( y, x );
   return fv;
} // minmax
```

ή την *pqer* (που είδαμε στις §14.9 και §15.10) ως εξής:

```
PairT< double, double > pqer( double x, double y, double z )
{
   if ( x == y || x == -y || z <= 0 )
      throw ApplicXptn( "pqer", ApplicXptn::paramErr, x, y, z );
   PairT< double, double > fv;
   fv.first = x*y*pow(z, x-y)/(x*x-y*y);
   fv.second = (x*y-1/x)/z;
   return fv;
} // pqer
```

Έτσι, έχουμε συναρτήσεις με τύπο που λέγαμε ότι είναι προτιμότερες από τις "`void`". Βεβαίως, αλλά στην περίπτωση αυτήν ο τύπος είναι κάπως «φτιαχτός»…

25.4 Κατανομή σε Αρχεία

Πριν προχωρήσουμε, ας κάνουμε μια επισήμανση, συμπλήρωμα στην §19.3: Το περίγραμμα μιας κλάσης γράφεται, κατ' αρχήν, σε ένα αρχείο, το "`.h`". Έτσι, για παράδειγμα, το *PairT* θα φυλαχτεί σε ένα αρχείο, το `PairT.h`, που θα έχει:

```
#ifndef _PAIRT_H
#define _PAIRT_H

using namespace std;

template < typename T1, typename T2 >
struct PairT
{
   T1 first;
   T2 second;

   PairT()
      :  first( T1() ), second( T2() ) { };
   PairT( const T1& x, const T2& y )
      :  first( x ), second( y ) { };
   template< typename U, typename V >
   PairT( const PairT<U, V>& p )
      : first( p.first ), second( p.second ) { };
}; // PairT

template < typename T1, typename T2 >
bool operator==( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first == y.first && x.second == y.second );  }

template < typename T1, typename T2 >
bool operator<( const PairT<T1, T2>& x, const PairT<T1, T2>& y )
{  return ( x.first < y.first ||
           (!(y.first < x.first) && x.second < y.second) );  }

template < typename T1, typename T2 >
PairT< T1, T2 > make_pair( T1 x, T2 y )
{  return PairT< T1, T2 >( x, y );  }

#endif // _PAIRT_H
```

25.5 Παράμετροι και Εξειδικεύσεις

Στην §14.7.1 λέγαμε ότι ένα περίγραμμα (συνάρτησης) «*μπορεί να έχει ως παραμέτρους*

- *έναν ή περισσότερους τύπους ή/και*
- *ακέραιους,*
- *βέλη ή αναφορές.*»

Ας τα δούμε με περισσότερες λεπτομέρειες. Οι παράμετροι ενός περιγράμματος μπορεί να είναι:

- Το λεξικό σύμβολο "`class`" ή "`typename`" ακολουθούμενο από όνομα της παραμέτρου. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμη (τύπο) της παραμέτρου.

  Τα "`class`" και "`typename`" δεν έχουν οποιαδήποτε νοηματική διαφορά. Μπορείς να χρησιμοποιείς όποιο θέλεις.[4]
- Το λεξικό σύμβολο "`template`" ακολουθούμενο από λίστα παραμέτρων περιγράμματος μέσα σε "`<`" και "`>`" ακολουθούμενη από το σύμβολο "`class`" και ένα όνομα. Μπορεί να ακολουθείται από "`=`" και ερήμην καθορισμένη τιμή (περίγραμμα) της παραμέτρου. Το παρακάτω παράδειγμα είναι από το πρότυπο C++03:

```
template< typename T > class myarray { /* ... */ };

template< typename K, typename V,
          template< typename T> class C = myarray >
class Map
```

[4] Πάντως το "`typename`" είναι πιο σωστό –παρ' όλο που ορισμένοι μεταγλωττιστές έχουν πρόβλημα με αυτό– αφού μπορείς εκεί να βάλεις "`int`" ή "`double`" (ή άλλα παρόμοια) που δεν είναι κλάσεις.

```
{
   C< K > key;
   C< V > value;
// ...
};
```

Όπως βλέπεις η ερήμην καθορισμένη τιμή "`myarray`" είναι περίγραμμα κλάσης της μορφής "`template< typename T > class C`".

- Παράμετρος ακέραιου ή απαριθμητού τύπου.
- Παράμετρος-βέλος προς αντικείμενο ή συνάρτηση.
- Παράμετρος-αναφοράς αντικειμένου ή συνάρτησης.
- Παράμετρος-βέλος προς μέλος.

Αν κάποιο στιγμιότυπο –ας πούμε το *BStringT<AClass>*– που βγαίνει αυτόματως δεν καλύπτει τις απαιτήσεις σου μπορείς, όπως και στα περιγράμματα συναρτήσεων, να κάνεις μια εξειδίκευση:

```
template<> class BStringT<AClass> { /* . . . */ };
```

Αλλά πρόσεχε:

♦ ***Για την εξειδίκευση θα πρέπει να ορίσεις από την αρχή τα πάντα.***

Η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!

Παρατήρηση:▶

Αυτό που θα λύνει το πρόβλημά σου σε πολλές περιπτώσεις είναι η εξειδίκευση μιας ή περισσότερων μεθόδων. Δηλαδή: αφού για κάθε μέθοδο γράφουμε ένα περίγραμμα συνάρτησης μπορούμε να κάνουμε εξειδίκευση του περιγράμματος της μεθόδου για τις περιπτώσεις που μας ενδιαφέρουν.

Έστω για παράδειγμα το περίγραμμα κλάσης:

```
template < typename K > class A
{
public:
   A( const K& b=0 ) {  a = b;  }
   void operator+=( const A& q );
   const K& getA() {  return a;  }
private:
   K a;
}; // template < typename K > class A
```

Για την περίπτωση που παίρνουμε στιγμιότυπο του *A* για `bool` –δηλαδή για την κλάση `A<bool>`– θέλουμε να έχουμε την πράξη "`||`" αντί για τη "`+`" και τη "`&&`" αντί για τη "`*`". Ο τελεστής που έχουμε εδώ θα πρέπει να γίνεται για την κλάση αυτήν "`||=`"(ας πούμε).

Μπορούμε λοιπόν να πούμε:

```
template < typename K >
void A<K>::operator+=( const A<K>& q )
{  a += q.a;  }
```

και να εξειδικεύσουμε:

```
template<>
void A<bool>::operator+=( const A<bool>& q )
{  a = a || q.a;  }
```

Αυτό είναι προτιμότερο από το να γράψουμε μια ολόκληρη κλάση:

```
template<> class A< bool > { /* . . . */ }; // ◀
```

Μια δυνατότητα που υπάρχει για περιγράμματα κλάσεων αλλά όχι για περιγράμματα συναρτήσεων είναι οι *προκαθορισμένες τιμές παραμέτρων του περιγράμματος*. Για παράδειγμα, μετά τον ορισμό:

```
template < typename T1=int, typename T2=char, int n=1 >
class C { /* . . . */ }; // C
```

μπορείς να δώσεις δηλώσεις όπως:

```
C<> ob0;
C< double > ob1;
C< long, int > ob2;
C< unsigned, wchar_t, 2 > ob3;
```

που είναι ισοδύναμες με:

```
C< int, char, 1 > ob0;
C< double, char, 1 > ob1;
C< long, int, 1 > ob2;
C< unsigned int, wchar_t, 2 > ob3;
```

Όπως είναι φυσικό, ισχύει αυτό που είπαμε στην §14.2 για παραμέτρους συναρτήσεων: «*Αν θέλεις να παραλείψεις κάποιο όρισμα, εκτός από το τελευταίο, θα πρέπει να παραλείψεις και όλα τα ορίσματα που το ακολουθούν.*» Δηλαδή: μη σκεφτείς να γράψεις "`C<3> x;`" επειδή βαριέσαι να γράψεις το σωστό: "`C<int, char, 3> x;`"

25.5.1 Μερική Εξειδίκευση Περιγράμματος Κλάσης

Όπως είδαμε, σε ένα περίγραμμα συνάρτησης μπορείς να επιφορτώσεις ένα ή περισσότερα περιγράμματα συναρτήσεων ή/και μια ή περισσότερες απλές συναρτήσεις. Επιφόρτωση κλάσεων ή περιγραμμάτων κλάσεων δεν υπάρχει αλλά στην εξειδίκευση περιγραμμάτων κλάσεων έχεις μια δυνατοτητα που δεν υπάρχει για την εξειδίκευση περιγραμμάτων συναρτήσεων: η **μερική εξειδίκευση** (partial specialization) *που δίνει περίγραμμα κλάσης και όχι κλάση.*

Δες μερικά παραδείγματα (από το πρότυπο C++03). Ας πούμε ότι έχουμε το πρωτεύον περίγραμμα:

```
template< typename T1, typename T2, int I > class A { /* . . . */ };
```

που έχει δύο παραμέτρους-τύπους και μια ακέραιη παράμετρο.

α) Μια εξειδίκευση είναι η εξής:

```
template< typename T1, typename T2, int I >
class A< T1*, T2, I > { /* . . . */ };
```

Εδώ περιορίζεται ο τύπος της πρώτης παραμέτρου του περιγράμματος: είναι τύπος βέλους.

β) Η εξειδίκευση:

```
template< typename T1, typename T2, int I >
class A< T1, T2*, I > { /* . . . */ };
```

είναι σαν την πρώτη αλλά θέλουμε τύπο βέλους στη δεύτερη παράμετρο.

γ) Μια άλλη εξειδίκευση είναι η εξής:

```
template< typename T, int I > class A< T, T*, I > { /* . . . */ };
```

όπου οι *T1* και *T2* δεν είναι άσχετες μεταξύ τους: η δεύτερη είναι βέλος προς αντικείμενο της πρώτης.

δ) Μια τέταρτη εξειδίκευση είναι η:

```
template< typename T > class A< int, T*, 5 > { /* . . . */ };
```

Εδώ καθορίζεται η πρώτη παράμετρος να είναι ο τύπος "`int`" και η τρίτη να έχει τιμή "`5`".

Θα πρέπει να επιστήσουμε την προσοχή σου στον ορισμό των μεθόδων ενός μερικώς εξειδικευμένου περιγράμματος κλάσης.

Όταν ζητήσουμε ένα στιγμιότυπο ενός περιγράμματος κλάσης τότε γίνεται το εξής: Ο μεταγλωττιστής προσπαθεί να ταιριάσει τις τιμές των παραμέτρων του στιγμιότυπου με αυτά των εξειδικεύσεων.

- Αν βρει μια μόνον εξειδίκευση που να ταιριάζει παράγει το στιγμιότυπο από αυτήν.
- Αν βρει δύο ή περισσότερες εξειδικεύσεις επιλέγει την πιο εξειδικευμένη. Αν δεν μπορεί να διαλέξει βγάζει λάθος.
- Αν δεν βρει εξειδίκευση που να ταιριάζει θα βγάλει το στιγμιότυπο από το αρχικό περίγραμμα.

Ας πούμε ότι έχουμε το περίγραμμα και τις εξειδικεύσεις που είδαμε παραπάνω και έχουμε την

```
A< int, int, 1 > a1;
```

Οι παράμετροι του στιγμιότυπου δεν ταιριάζουν με οποιαδήποτε από τις εξειδικεύσεις. Έτσι, θα παραχθεί από το αρχικό περίγραμμα βάζοντας "`int`" στους *T1*, *T2* και "`1`" στην *I*.

Ας δούμε τώρα τη δήλωση:

```
A< int, int*, 1 > a2;
```

Οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στους *T1* και *T2* και "`1`" στην *I*.
- Με την εξειδίκευση (γ) αν βάλουμε "`int`" στον *T* και "`1`" στην *I*.

Η (γ) είναι πιο εξειδικευμένη και από αυτήν θα παραχθεί το στιγμιότυπο.

Για τη δήλωση:

```
A< int, char*, 5 > a3;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν

- Με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`5`" στην *I*.
- Με την εξειδίκευση (δ) αν βάλουμε "`char`" στον *T*.

Το στιγμιότυπο θα παραχθεί από τη (δ) που είναι πιο εξειδικευμένη.

Για τη δήλωση:

```
A< int, char*, 1 > a4;
```

οι παράμετροι του στιγμιότυπου ταιριάζουν μόνο με την εξειδίκευση (β) αν βάλουμε "`int`" στον *T1*, "`char`"και *T2* και "`1`" στην *I*.

Τέλος, για τη δήλωση:

```
A< int*, int*, 2 > a5;
```

οι παράμετροι ταιριάζουν με την (α) και τη (β) που όμως είναι το ίδιο εξειδικευμένες. Εδώ ο μεταγλωττιστής, μη μπορώντας να δημιουργήσει τον τύπο της *a5*, θα βγάλει λάθος.

Το πρόγραμμα που ακολουθεί δοκιμάζει τα παραπάνω:

```
#include <iostream>

using namespace std;

template< typename T1, typename T2, int I >
struct A
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1, T2, I >::f()  {   cout << "f 0" << endl;  }

template<> void A< int, int, 0 >::f()
{   cout << "f 0 ovrld" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1*, T2, I >
{
   void f();
};

template< typename T1, typename T2, int I >
void A< T1*, T2, I >::f()  {  cout << "f a" << endl;  }

template< typename T1, typename T2, int I >
struct A< T1, T2*, I >
{
   void f();
};
```

```
template< typename T1, typename T2, int I >
void A< T1, T2*, I >::f()  {  cout << "f b" << endl;  }

template< typename T, int I >
struct A< T, T*, I >
{
   void f();
};

template< typename T, int I >
void A< T, T*, I >::f()  {  cout << "f c" << endl;  }

template< typename T >
struct A< int, T*, 5 >
{
   void f();
};

template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }

int main()
{
   A< int, int, 0 > a0;
   A< int, int, 1 > a1;
   A< int, int*, 1 > a2;
   A< int, char*, 5 > a3;
   A< int, char*, 1 > a4;
   A< int*, int*, 2 > a5;

   a0.f();
   a1.f();
   a2.f();
   a3.f();
   a4.f();
   a5.f();
}
```

Για την "`A< int*, int*, 2 > a5;`" ο μεταγλωττιστής[5] θα μας βγάλει λάθος: "`Too many candidate template specializations from 'A<T1,T2,2>' in function main()`". Αν διαγράψουμε αυτήν τη δήλωση καθώς και την εντολή "`a5.f();`" όλα πάνε καλά και παίρνουμε αποτέλεσμα:

```
f 0 ovrld
f 0
f c
f d
f b
```

Τώρα κάνουμε την εξής δοκιμή: αφαιρούμε την

```
template< typename T >
void A< int, T*, 5 >::f()  {  cout << "f d" << endl;  }
```

Αυτήν τη φορά θα διαμαρτυρηθεί ο συνδέτης: "`Error: Unresolved external 'A<int, char *, 5>::f()' referenced from …`" Όπως λέγαμε πιο πριν «*η εξειδίκευση δεν κληρονομεί οτιδήποτε από το περίγραμμα!*» Το ίδιο ισχύει και για τα περιγράμματα που προκύπτουν από μερικές εξειδικεύσεις ενός πρωτεύοντος περιγράμματος: *δεν κληρονομούν ότιδήποτε από αυτό.*

[5] Borland C++ v.5.5.

25.6 Περιγράμματα και Κληρονομιές

Τι δυνατότητες κληρονομιάς έχουν τα περιγράμματα κλάσεων; Ό,τι μπορείς να σκεφτείς.

25.6.1 Κληρονομιά Κλάσης από Περίγραμμα Κλάσης

Μια κλάση μπορεί να κληρονομεί *στιγμιότυπο* περιγράμματος κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

class D :  public B< char >
{
public:
   D( char a1=0, int a2=0 )
      :  B<char>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<char>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Πρόσεξε την προετοιμασία που κάναμε στη βασική κλάση (περίγραμμα) για να μπορεί να κληρονομηθεί: έχουμε δηλώσει **virtual** τον καταστροφέα. Ακόμη κάναμε το ίδιο και στη μέθοδο *display()* αφού περιμένουμε –σε περίπτωση κληρονομιάς– να υπάρξει μέθοδος της παράγωγης κλάσης με το ίδιο όνομα που θα πρέπει να υπερισχύσει.

25.6.2 Κληρονομιά Περιγράμματος Κλάσης από Κλάση

Μερικές φορές μπορεί να θέλεις να χρησιμοποιήσεις –σε ένα περίγραμμα που γράφεις– εργαλεία που έχεις σε κάποια κλάση. Ένας τρόπος είναι και η κληρονομια. Για παράδειγμα:

```
class B
{
public:
   B( int a=0 ) :  bCount( a ) { };
   virtual ~B() { };
   unsigned int getCount() const {  return bCount;  }
// . . .
protected:
   unsigned int bCount;
}; // B

template < typename C, size_type sz=10 >
class D :  public B
{
public:
   D( int a=sz )
   {  if ( a <= 0 )  throw a;
      dArr = new C[a];   bCount = a;  };
   virtual ~D() {  delete[] dArr;  };
// . . .
private:
   C* dArr;
}; // D
```

25.6.3 Κληρονομιά Περιγράμματος Κλάσης από Περίγραμμα

Η πιο ενδιαφέρουσα από τις τρεις περιπτώσεις είναι η κληρονομιά περιγράμματος κλάσης από άλλο περίγραμμα κλάσης. Για παράδειγμα:

```
template < typename T >
class B
{
public:
   B( T a ) :  mb( a ) { };
   virtual ~B() { };
   virtual void display( ostream& tout )  {  tout << mb;  };
private:
   T mb;
}; // B

template < typename C >
class D :  public B< C >
{
public:
   D( C a1=0, int a2=0 )
      :  B<C>( a1 ), md( a2 ) {  };
   virtual ~D() { };
   virtual void display( ostream& tout )
   {  B<C>::display( tout );  tout << "  " << md;  }
private:
   int md;
}; // D
```

Όπως βλέπεις και εδώ προετοιμάζουμε για κληρονομιά το περίγραμμα κλάσης *B*, από το οποίο θα προέλθει η βασική κλάση. Από αυτά που έχουμε μέχρι εδώ δεν έχει ορισθεί κάποια κλάση.

Κλάση δημιουργείται όταν εκτελεσθεί η

```
   D<char> x( 'x', 7 );
```

όπου ζητείται κλάση-στιγμιότυπο του *D* για τιμή παραμέτρου "`char`". Πριν από αυτήν θα πρέπει να δημιουργηθεί η `B<char>` την οποία θα κληρονομήσει η `D<char>`.

Ας δούμε και ένα πολύ απλό παράδειγμα από την STL. Για να μπορούμε να περάσουμε μια συνάρτηση (κατηγόρημα) θα πρέπει να τη «μετατρέψουμε» σε κλάση· για την ακρίβεια σε συναρτησοειδές (§22.6.2). Η C++ έχει έτοιμα μερικά περιγράμματα τέτοιων συναρτησοειδών και μπορείς να τα χρησιμοποιήσεις βάζοντας στο πρόγραμμά σου "`#include <functional>`". Εκεί μπορείς να βρεις τα εξής περιγράμματα (μεταξύ άλλων):

```
template < typename Arg1, typename Arg2, typename Result >
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};


template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Το *binary_function* είναι ένα «άδειο» περίγραμμα ενώ το δεύτερο είναι περίγραμμα συναρτησοειδούς. Μέσα στο πρόγραμμά σου μπορείς να γράψεις:

```
#include <functional>
// . . .
using namespace std;
// . . .
   less<int> lt;
```

Η "`less<int>`" είναι κλάση (συναρτησοειδές) και η "`lt(1,-1)`" θα δώσει τιμή "`false`".

Δες και μιαν άλλην περίπτωση:

```
template < typename C1, typename C2 >
class D1 :  public B< C1 >
{
public:
   D1( C1 a1=0, C2 a2=0 )
      :  B<C1>( a1 ), md( a2 ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B<C1>::display( tout );  tout << "  " << md;  }
private:
   C2 md;
}; // D1
```

Εδώ, η πρώτη παράμετρος (*C1*) καθορίζει το στιγμιότυπο της βασικής το οποίο θα κληρονομήσει η παράγωγη ενώ η δεύτερη (*C2*) αναφέρεται μόνον στην παράγωγη.

Τέλος, θα αναφέρουμε δύο πιθανές δυσάρεστες εκπλήξεις που σε περιμένουν στην περίπτωση κληρονομιάς περιγράμματος από περίγραμμα.

1. Ας πούμε ότι στο «βασικό περίγραμμα» *B* έχεις μια συνάρτηση-μέλος

```
template < typename T >
class B
{
public:
   void f() {  /* . . . */  }
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* καλεί την *f()*:

```
   void g() { /* . . . */ f(); /* . . . */ }
```

Είναι πιθανό ότι ο μεταγλωττιστής σου (π.χ. gcc) δεν θα το επιτρέψει.

Πώς θα τον πείσουμε να το δεχτεί; Γράφοντας την κλήση της *f()* ως "`this->f()`" ή "`B<C>::f()`".

2. Ας πούμε ότι στο «βασικό περίγραμμα» *B* ορίζεις έναν τοπικό δικό σου τύπο, π.χ. μια κλάση:

```
template < typename T >
class B
{
public:
   struct Qaz {  /* . . . */  };
// . . .
```

που δεν εξαρτάται από την παράμετρο *T* ενώ σε κάποια συνάρτηση του «παράγωγου περιγράμματος» *D* δηλώνεις μια μεταβλητή τύπου *Qaz*:

```
   void g() { Qaz x; /* . . . */ }
```

Και αυτό μπορεί να απορριφθεί από τον μεταγλωττιστή σου.

Πώς διορθώνεται; Έτσι:

```
   void g() { typename B<C>::Qaz x; /* . . . */ }
```

Για περισσότερα παραπέμπουμε στο (Cline 1999), §35.18 και 35.19.

25.7 Περιέχουσες Κλάσεις

Περιέχουσα κλάση (container class) είναι *περίγραμμα* για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων. **Περιέχον** (container) είναι ένα αντικείμενο στιγμιοτύπου περιέχουσας κλάσης. Παραδείγματα:

- Η κλάση *SList* που είναι συλλογή αντικειμένων κλάσης *GrElmn* (§21.11) θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.

- Οι κλάσεις *CourseCollection*, *StudentCollection* και *StudentInCourseCollection* –που είδαμε στα Project 4 και 6 και είναι συλλογές αντικειμένων κλάσης *Course*, *Student* και *StudentInCourse* αντιστοίχως– θα μπορούσαν να είναι στιγμιότυπα μιας τέτοιας κλάσης.[6]
- Όλοι οι πίνακες που έχουμε δει στα παραδείγματά μας είναι συλλογές αντικειμένων του ίδιου τύπου. Βέβαια είναι λιγο δύσκολο να τους δούμε ως στιγμιότυπα ενός περιγράμματος κλάσης.

Τώρα θα πεις: Ένα αντικείμενο κλάσης *CourseCollection* περιέχει πίνακα με (βέλη προς) αντικείμενα κλάσης *Course*. Τι έχουμε εδώ; Περιέχον μέσα σε περιέχον; Όχι! Το περιέχον είναι ο πίνακας. Το αντικείμενο που τον περιέχει είναι το εργαλείο μας για να κάνουμε τον χειρισμό του πίνακα σύμφωνα με ορισμένους κανόνες (που προκύπτουν από την αναλλοίωτη). Λέμε ότι η κλάση *CourseCollection* (ακριβέστερα: το περίγραμμα κλάσης του οποίου στιγμιότυπο θα ήταν η *CourseCollection*) είναι **προσαρμογέας περιέχοντος** (container adaptor) ή **περιέχον-προσαρμογέας**. Αυτά ξεκαθαρίζουν στο παρακάτω

Παράδειγμα – Μια στοίβα σε πίνακα

Με τις

```
K   arr[sz];
int top;
```

και με δύο συναρτήσεις –*push()* και *pop()*– μπορούμε να υλοποιήσουμε μια **στοίβα** ((LIFO) stack) με στοιχεία τύπου *K*. Σε κάθε στιγμιότυπο όπου καθορίζεται ο τύπος *K* (π.χ. `int` ή *Date* ή *Course* κλπ) ο πίνακας είναι μια συλλογή αντικειμένων αυτού του τύπου. Έχουμε λοιπόν ένα περιέχον.

Αλλά το περιέχον μας έχει πρόβλημα: Πολλά μπορούμε να κάνουμε με τα στοιχεία ενός πίνακα! Έτσι:

- Σε μια στοίβα η θέση της κορυφής (*top*) μπορεί να μεταβάλλεται μόνον από τις *push()* και *pop()* κατά "`+1`" ή "`-1`" αντιστοίχως. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`top = 47`" ή "`top += 19`".
- Ακόμη, σε μια στοίβα μπορούμε να «δούμε» μόνο το στοιχείο που δείχνει η κορυφή. Εδώ τίποτε δεν μας εμποδίζει να βάλουμε εντολές όπως "`arr[47] = ...`" ή "`q = arr[29]`".

Για να εμποδίσουμε τέτοια λάθη «κρύβουμε» τα `arr` και `top` μέσα σε έναν προσαρμογέα περιέχοντος:

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
// . . .
   void push( const K& v );
   void pop();
   const K& getTop() const;
// . . .
private:
   K   arr[sz];
   int top;
}; // StackT
```

Ολόκληρο το περίγραμμα μπορεί να είναι:[7]

```
template < typename K, size_type sz = 100 >
class StackT
{
public:
   StackT() { top = -1; }  // empty stack
```

[6] Θα το δούμε σε επόμενα Project.

[7] Το κατηγόρημα για την άδεια στοίβα θα το βαφτίζαμε *isEmpty()*, αλλά προτιμούμε να ακολουθήσουμε την ονοματολογία της STL.

```
   StackT( const StackT& aSt );
   void push( const K& v );
   void pop();
   const K& getTop() const;
   bool empty() const {  return ( top == -1 );  }
   bool isFull() const {  return ( top == sz-1 );  }
private:
   K   arr[sz];
   int top;
}; // StackT

struct StackTXptn :  public exception
{
   static const int fnLength = 100;
   enum { stackFull, stackEmpty };
   char funcName[fnLength];
   int  errorCode;

   StackTXptn( const char* fn, int ec ) :  errorCode(ec)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
}; // StackTXptn

template < typename K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   if ( top == sz-1 )
      throw StackTXptn( "push", StackTXptn::stackFull );
   ++top;  arr[top] = v;
} // push

template < typename K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
} // pop

template < typename K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( top == -1 )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return arr[top];
} // getTop
```

Το περίγραμμα είναι πολύ απλό, αλλά όχι και τόσο χρήσιμο. Πρόβλημα; Το σταθερό μέγεθος του πίνακα. Μπορούμε να τον κάνουμε δυναμικό και να χρησιμοποιήσουμε τη *renew()*. Αλλά, αν πρέπει να κάνουμε κάθε τόσο αντιγραφές ολόκληρου του πίνακα οι καθυστερήσεις θα είναι μεγάλες.

Να και ένα προγραμματάκι που δοκιμάζει την παραπάνω υλοποίηση:

```
int main()
{
   StackT<int> intStack;

   cout << "Pushing integers on intStack" << endl;
   for ( int i = 0; i < 5; i++ )
   {
      intStack.push( i );       // put items in the stack
      cout << i << ' ';
   }
   cout << endl;

   cout << endl << "Popping integers from intStack" << endl;
   while ( !intStack.empty() )
   {
```

```
      cout << intStack.getTop() << ' ';
      intStack.pop();                    // remove items
   }
   cout << endl;

   StackT<char> charStack;
   cout << endl << endl
        << "Pushing characters on charStack" << endl;
   for ( char c = 'A'; c < 'E'; c++ )
   {
      charStack.push( c );      // put items in the stack
      cout << c << ' ';
   }
   cout << endl;

   cout << endl << "Popping characters from charStack" << endl;
   while ( !charStack.empty() )
   {
      cout << charStack.getTop() << ' ';
      charStack.pop();   // remove items
   }
   cout << endl;
} // main
```

Αυτή η στοίβα ήταν το «ορεκτικό». Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.

Παρατήρηση: ▶

Όλος ο κόσμος θα έγραφε ένα περίγραμμα σαν

```
template < typename K, size_type sz >
const K& StackT<K,sz>::pop()
{
   if ( top == -1 )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   --top;
   return arr[top+1];
} // pop
```

Γιατί εδώ γράψαμε δύο χωριστές μεθόδους-περιγράμματα; Για ασφάλεια στις εξαιρέσεις!

Ας εξηγηθούμε: έστω ότι η *v* είναι μεταβλητή κλάσης *K*, το *aStack* είναι αντικείμενο κλάσης `StackT<K>` και ρίχνεται εξαίρεση κατά την εκτέλεση της:

```
v = aStack.pop();
```

Έστω ότι η εκχώρηση στην κλάση *K* έχει ισχυρή εγγύηση ασφάλειας πράγμα που σημαίνει ότι η τιμή της *v* δεν θα αλλάξει. Αλλά τι γίνεται με το *aStack*; Η τιμή που αφαιρέθηκε από την κορυφή του χάθηκε!

Με την υλοποίηση που δώσαμε η ίδια δουλειά θα γίνει με τις:

```
v = aStack.getTop();
aStack.pop();
```

Αν κατά την εκτέλεση της πρώτης εντολής ριχτεί εξαίρεση η δεύτερη εντολή δεν θα εκτελεσθεί.

Πάντως αν το πρόγραμμά σου έχει σχεδιαστεί για πολυνηματική εκτέλεση και δύο ή περισσότερα νήματα παίρνουν τιμές από το *aStack* υπάρχει άλλο πρόβλημα. Οι δύο λειτουργίες θα πρέπει να γίνονται από *μια* συνάρτηση που θα πρέπει να έχει και **ατομικότητα** (atomicity), που με απλά λόγια σημαίνει ότι εκτελείται σαν *μια εντολή* (πριν τελειώσει η εκτέλεσή της δεν μπορεί να αλλάξει νήμα εκτέλεσης). Η C++11, αλλά και οι API των ΛΣ, σου δίνουν εργαλεία για να λύσεις παρόμοια προβλήματα.◀

❖❖❖

25.7.1 Το Περίγραμμα μιας Λίστας

Πιο πάνω είπαμε «*Η κλάση SList, που είναι συλλογή αντικειμένων κλάσης GrElmn (§21.11), θα μπορούσε να είναι στιγμιότυπο ενός τέτοιου περιγράμματος. Θα το δούμε στη συνέχεια.*» Ας το δούμε λοιπόν. Θα ξεκινήσουμε από την τελευταία μορφή της *SList*, όπως την είδαμε στην §21.11 και στην §22.9 όπου κάναμε εισαγωγή των προσεγγιστών.

Όπως στη *BString* αντικαταστήσαμε το "`char`" με "*K*" εδώ θα αντικαταστήσουμε το "*GrElmn*" με "*K*". Πριν από αυτό όμως χρειάζονται μερικές αλλαγές. Οι μέθοδοι *find1Elmn()*, *get1Elmn()* και *delete1Elmn()* έχουν μια παράμετρο τύπου `int` αφού περιμένουν εκεί ένα όρισμα, τον ατομικό αριθμό, που είναι και το κλειδί για τη *GrElmn*. Αν τις αφήσουμε όπως είναι το περίγραμμα θα μπορεί να δώσει μόνον στιγμιότυπα-συλλογές αντικειμένων με κλειδί κάποιον ακέραιο. Φυσικά δεν θέλουμε κάτι τέτοιο και τις αλλάζουμε σε:

```
bool SList::find1Elmn( const GrElmn& aDtItem ) const
{
   return ( findPtr(aDtItem) != slTail );
} // SList::find1Elmn

const GrElmn& SList::get1Elmn( const GrElmn& aDtItem ) const
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::get1Elmn

void SList::delete1Elmn( const GrElmn& aDtItem )
{
   ListNode* ptrToNode( findPtr(aDtItem) );
// . . .
} // SList::delete1Elmn
```

Στη συνέχεια θα ξαναδούμε αυτές τις μεθόδους.

Παρατήρηση: ▶

Τουλάχιστον η *get1Elmn()* φαίνεται κάπως περίεργη: Ζητούμε να μας επιστρέψει κάτι που το δίνουμε ως όρισμα όταν την καλούμε; Όχι βέβαια. Η κλήση της θα γίνεται συνήθως ως εξής:

```
tmp = lst.get1Elmn( GrElmn(aa) );
```

Δηλαδή στο *aDtItem* θα περνάει ένα αντικείμενο που θα έχει μόνο το κλειδί. Για τα υπόλοιπα θα στηριζόμαστε στον ορισμό της ισότητας (`operator==()`) που έχουμε για τη *GrElmn*.◀

Κάνοντας τώρα την αντικατάσταση του "*GrElmn*" με "*K*" έχουμε:

```
template < typename K >
class SListT
{
private:
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      K& operator*();
      K* operator->();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
```

```
      ListNode* iPNode;
   }; // Iterator

   SListT();
   SListT( const SListT& rhs );
   ~SListT();
   SListT& operator=( const SListT& rhs );
   void swap( SListT& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
   bool find1Elmn( const K& aDtItem ) const;
   const K& get1Elmn( const K& aDtItem ) const;
   void insert1Elmn( const K& aDtItem );
   void delete1Elmn( const K& aDtItem );
   void save( ostream& bout,
              void (*wrProc)(const K&, ostream&) ) const;
   void display( ostream& tout ) const;
private:
   ListNode* slHead;
   ListNode* slTail;

   ListNode* findPtr( const K& aDtItem ) const;
   void push_front( const K& aData );
}; // SList
```

και η αντίστοιχη κλάση εξαιρέσεων:

```
struct SListTXptn
{
   enum { allocFailed, notFound, listEnd };
   char  funcName[100];
   int   errorCode;
   int   errIntVal;
   SListTXptn( const char* fn, int ec, int iv=0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errorCode = ec;  errIntVal = iv;  }
}; // SListXptn
```

Ας δούμε τα περιγράμματα συναρτήσεων που επιφορτώνουν τους τελεστές "++", "*" και "->" για τον *SListT<K>::Iterator*:

```
template < typename K >
SListT<K>::Iterator& SListT<K>::Iterator::operator++()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator++",
                        SListTXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SListT<K>::Iterator::operator++

template < typename K >
K& SListT<K>::Iterator::operator*()
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListTXptn( "Iterator::operator*",
                        SListTXptn::listEnd );
   return ( iPNode->lnData );
} // SListT<K>::Iterator::operator*

template < typename K >
K* SListT<K>::Iterator::operator->() const
{
   if ( iPNode == slTail ) // ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator->",
                       SListXptn::listEnd );
   return ( &(iPNode->lnData) );
} // SList::Iterator::operator->
```

Στη συνέχεια θα δούμε μερικές από τις μεθόδους του περιγράμματος ξεκινώντας με την

```
template < typename K >
void SListT<K>::push_front( const K& aData )
{
   ListNode* pnln;
   try {  pnln = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListTXptn( "push_front", SListTXptn::allocFailed );  }
   pnln->lnData = aData;  pnln->lnNext = slHead;
   slHead = pnln;
} // SList::push_front
```

Εδώ τώρα πρόσεξε τα εξής:

- Για να εκτελεσθεί η "`new ListNode`" θα πρέπει να δημιουργηθεί και το μέλος *lnData* από τον ερήμην δημιουργό της *K*.
- Εκτέλεση της "`pnln->lnData = aData`" σημαίνει εκτέλεση του (αντιγραφικού) τελεστη εκχώρησης της *K*.

Επομένως:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***

Βέβαια θα πεις ότι θα μπορούσαμε να αποφύγουμε τον ερήμην δημιουργό και τον τελεστή εκχώρησης αν εφοδιάσουμε την *ListNode* με δημιουργούς:

```
   struct ListNode
   {
      K         lnData;
      ListNode* lnNext;
      ListNode()
         : lnNext( 0 ) { };
      explicit ListNode( const K& aData )
         : lnData( aData ), lnNext( 0 ) { };
   }; // ListNode
```

και αντί για:

```
   pnln = new ListNode;  pnln->lnData = aData;
```

να γράφουμε

```
   pnln = new ListNode( aData );
```

Εδώ καλείται ο δεύτερος δημιουργός που –με τη σειρά του– θα καλέσει τον δημιουργό αντιγραφής (`lnData(aData)`) της *K*.

Σωστό! Αλλά πρόσεξε το εξής: Εμείς κάνουμε αυτές τις επισημάνσεις για να κατάλάβεις τις απαιτήσεις που (θα δούμε ότι) βάζει η STL για τις κλάσεις που πιθανόν θα θελήσεις να βάλεις σε κάποιο από τα περιέχοντά της. Εκεί δεν θα έχεις επιλογή για το πώς θα γράψεις το περιέχον· είναι έτοιμο. Έτσι, η παρατήρηση που κάναμε πιο πάνω όχι μόνον δεν αναιρεί τους περιορισμούς που είχαμε αλλά βάζει έναν ακόμη:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***

Ας σκεφτούμε τώρα λιγάκι τη *findPtr()*. Κατ' αρχήν θα μπορούσαμε να γράψουμε το περίγραμμα:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   slTail->lnData = aDtItem;
   fv = slHead;
   while ( (fv->lnData) != aDtItem )  fv = fv->lnNext;
   return fv;
} // SListT<K>::findPtr
```

αλλά εκείνη η "`slTail->lnData = aDtItem`" –αντιγραφή ολόκληρου αντικειμενου κλάσης *K*–[8] έχει πρόβλημα. Αν μπορούμε να θεωρήσουμε ότι ένα αντικείμενο *GrElmn* είναι μικρό και η αντιγραφή του δεν είναι χρονοβόρα δεν μπορούμε να πούμε το ίδιο πράγμα για το αντικείμενο οποιασδήποτε κλάσης *K* θα κληθεί να φιλοξενήσει το *SListT*.

Εγκαταλείπουμε λοιπόν την αντιγραφή στον φρουρό και επιστρέφουμε στις διπλές συγκρίσεις:

```
template < typename K >
SListT<K>::ListNode* SListT<K>::findPtr( const K& aDtItem ) const
{
   ListNode* fv;
   for ( fv = slHead; fv != slTail; fv = fv->lnNext )
      if ( fv->lnData == aDtItem ) break;
   return fv;
} // SListT<K>::findPtr
```

Αυτό το περίγραμμα όμως είναι σαν να μας λέει: «Άλλαξε τα βέλη, σε προσεγγιστές!» Ας το κάνουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::findIt( const K& aDtItem ) const
{
   Iterator fv;
   for ( fv = begin(); fv != end(); ++fv )
      if ( *fv == aDtItem ) break;
   return fv;
} // SListT<K>::findIt
```

Αν χρησιμοποιήσουμε τη νέα *findIt()* στη *get1Elmn()* θα έχουμε:

```
template < typename K >
const K& SListT<K>::get1Elmn( const K& aDtItem ) const
{
   Iterator itToNode( findIt(aDtItem) );
   if ( itToNode == end() )
      throw SListTXptn( "get1Elmn", SListTXptn::notFound,
                        aDtItem.getANumber() );
   return *itToNode;
} // SList<K>::get1Elmn
```

Εδώ όμως η *findIt()* δεν μας χρειάζεται· μπορούμε να χρησιμοποιήσουμε την *std::find()* –που είδαμε στην §22.9– και να γράψουμε:

```
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
```

Μπορείς να τροποποιήσεις με τον ίδιο τρόπο και τις *insert1Elmn()* και *delete1Elmn()*. Για την τελευταία θα χρειαστείς κάτι ακόμη: Στην περιοχή "`private`" της *Iterator* ορίζουμε:

```
// . . .
   private:
      ListNode* iPNode;
      ListNode* getPNode() const {  return iPNode;  }
   }; // Iterator
```

Έτσι, στην αρχή της *delete1Elmn()* θα έχουμε:

```
template < typename K >
void SListT<K>::delete1Elmn( const K& aDtItem )
{
   Iterator itToNode( std::find(begin(), end(), aDtItem) );
   if ( itToNode != end() ) // υπάρχει
   {
      ListNode* ptrToNode( itToNode.getPNode() );
      ListNode* pnln( ptrToNode->lnNext );
   // . . .
```

[8] Ενώ χρειαζόμαστε μόνο το κλειδί!

Παρατήρηση:▶
Σχετικώς με τον φρουρό: οι αντιγραφές δεν είναι το μόνο πρόβλημα. Αν τα αντικείμενα της περιεχόμενης κλάσης είναι μεγάλα η ύπαρξη του φρουρού είναι και σπατάλη μνήμης.◀

25.7.1.1 Ο Καταστροφέας

Τέλος, ας πούμε και δυο λόγια για τον καταστροφέα. Το μόνο πράγμα που άλλαζει είναι η επικεφαλίδα από "`SList::~SList()`" σε "`template <typename K> SListT<K>::~SListT()`". Υπάρχει όμως ένα άλλο ερώτημα σχετικώς με τη δήλωση: Μήπως θα έπρεπε να τον δηλώσουμε "`virtual`" ώστε να υπάρχει δυνατότητα να κληρονομηθούν οι κλάσεις-στιγμιότυπα του περιγράμματος; Αυτό είναι ένα ερώτημα που τίθεται από πολλούς για τα περιέχοντα της SLT και μάλιστα με κάπως διαφορετικό τρόπο: πώς «ξέφυγε» μια τόσο σοβαρή παράλειψη και οι καταστροφείς των περιεχόντων της STL δεν είναι "`virtual`";

Φυσικά δεν πρόκειται για λάθος ούτε για παράλειψη και ας το σκεφτούμε. Στη συνέχεια θα δούμε υλοποίηση στοίβας με μια λίστα. Για να δηλώσουμε μια στοίβα ακεραίων μπορούμε να κάνουμε ένα από τα εξής:

- Να ορίσουμε το περίγραμμα:

```
template < typename K >
class StackT
{
public:
// . . .
private:
   SListT<K> cont;
}; // StackT
```

και στη συνέχεια να δηλώσουμε:

```
StackT< int > aStack;
```

- Να ορίσουμε:

```
class IntStack : public SListT<int>  { /*. . . */ }
```

και στη συνέχεια:

```
IntStack aStack;
```

Τι θα πρέπει να προτιμήσουμε; Για να απαντήσουμε θα πρέπει να καταλάβουμε το νόημα της κάθε επιλογής:

- Στην πρώτη περίπτωση έχουμε το περίγραμμα όπως το είχαμε αρχικώς (εκτός από τη δεύτερη παράμετρο "`size_type sz = 100`") όπου αλλάξαμε το περιέχον από πίνακα σε λίστα.
- Στη δεύτερη περίπτωση λέμε ότι μια στοίβα ακεραίων είναι μια (`is_a`) λίστα ακεραίων.

Η δεύτερη επιλογή έρχεται σε αντίθεση με αυτό που λέγαμε στην §23.14: «*οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και τη σωστή νοηματική σχέση.*» Η στοίβα *μπορεί να υλοποιηθεί* με μια λίστα. Αλλά *δεν είναι* λίστα· ήδη την έχουμε υλοποιήσει με πίνακα.[9]

Στις περισσότερες περιπτώσεις που θα πάει το μυαλό σου σε κληρονομιά από περιέχον ένα σκεπτικό σαν το παραπάνω έχει νόημα. Δεν βάζουμε λοιπόν το "`virtual`" για να αποθαρρύνουμε τέτοιες σκέψεις.

Δηλαδή η κληρονομιά από το περίγραμμα λίστα δεν έχει νόημα ποτέ; Θα είχε νόημα αν κάναμε μια νέα δομή δεδομένων που θα ήταν ειδική περίπτωση της λίστας με απλή σύνδεση.

---

[9] Με την ορολογία της §23.15: αυτά είναι επιχειρήματα υπέρ της σχέσης "`has_a`"· επομένως και υπέρ της κληρονομιάς "`private`".

25.7.2 Η Περιεχόμενη Κλάση

Να καταγράψουμε απολογιστικά τις απαιτήσεις για την περιεχόμενη κλάση που βρήκαμε από το παράδειγμά μας μέχρι τώρα:

- ***Πρώτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Δεύτερη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει δημιουργό αντιγραφής.***
- ***Τρίτη Απαίτηση: Για τον περιεχόμενο τύπο K πρέπει να έχει ορισθεί ο τελεστής εκχώρησης.***

Φυσικά, χωρίς να καταγράψουμε, χρησιμοποιήσαμε το γεγονός ότι:

- ***Τέταρτη Απαίτηση: Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

Αυτές οι απαιτήσεις τίθενται για όλα τα περιέχοντα της STL. Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Αν, ας πούμε, θέλεις να κάνεις αναζητήσεις τιμών θα πρέπει να έχεις τη δυνατότητα για συγκρίσεις.

25.7.3 Ένα Πρόβλημα, μια Λύση και ένα Άλλο Πρόβλημα

Το περίγραμμα *slist*[10] (αλλά και το *list* της STL) έχει μέθοδο:

```
iterator erase( iterator pos )
```

Αν *a* είναι αντικείμενο κλάσης *slist*<*K*> και *p* ένας *slist*<*K*>::*iterator* τότε η εντολή

```
a.erase( p );
```

καταστρέφει τον κόμβο-στόχο του *p*. Το μήκος της λίστας μειώνεται κατά 1 χωρίς να αλλάζουν οι σχετικές θέσεις των άλλων κόμβων. Η μέθοδος επιστρέφει προσεγγιστή με στόχο το στοιχείο της λίστας που ακολουθεί αυτό που διαγράφηκε.

Αν θέλουμε να γράψουμε και εμείς μια μέθοδο

```
Iterator erase( Iterator pos );
```

για το *SlistT* θα αντιμετωπίσουμε το εξής πρόβλημα: «*θα πρέπει να αλλάξουμε την τιμή του lnNext του προηγούμενου κόμβου που δεν ξέρουμε ποιος είναι!*» Στην §21.11 δώσαμε μια λύση –που φαίνεται στο Σχ. 21-1– γράφοντας την *SList*::*erase1Elmn*. Στην περίπτωσή μας θα έχουμε:

```
template < typename K >
SListT<K>::Iterator SListT<K>::erase( SListT<K>::Iterator pos )
{
   if ( pos == end() )
      throw SListTXptn( "erase", SListTXptn::listEnd );
   ListNode* ptrToNode( pos.getPNode() );
   ListNode* pnln( ptrToNode->lnNext );
   ptrToNode->lnNext = pnln->lnNext;
   if ( pnln == slTail ) slTail = ptrToNode;
                    else ptrToNode->lnData = pnln->lnData;
   delete pnln;
   return pos;
} // SListT<K>::erase
```

Όπως βλέπεις, θα χρειαστούμε μια μέθοδο (`private`) της κλάσης *Iterator* που θα μας δίνει το βέλος που κρύβει ο προσεγγιστής:

```
      ListNode* getPNode() const {  return iPNode;  }
```

Φαίνεται λοιπόν ότι λύσαμε το πρόβλημά μας αλλά πρόσεξε τα προβλήματα της παραπάνω λύσης:

---

[10] της SGI. Στο C++11 προδιαγράφεται περιέχουσα κλάση με το όνομα "`forward_list`" και είναι λίστα με απλή σύνδεση. Η *forward_list* δεν έχει *erase()* με τα παραπάνω χαρακτηριστικά. Έχει μέθοδο *erase_after()* που καταστρέφει τον κόμβο που ακολουθεί τον κόμβο-στόχο.

- Που δείχνει ο `pos`; Στον κόμβο που έδειχνε αλλά που τώρα έχει ως περιεχόμενο αυτό του κόμβου που ακολουθούσε αυτόν που διαγράψαμε! Δηλαδή το `*pos` αλλάζει με τη διαγραφή. Ναι, αλλά ο `pos` είναι παράμετρος τιμής. Αυτή είναι μια περίεργη κατάσταση.
  Αν χρησιμοποιήσεις άλλη μέθοδο και ανακυκλώσεις ακριβώς τον κόμβο-στόχο του `pos` τότε τα πράγματα είναι χειρότερα.
- Αν δεν έχουμε φτάσει στον φρουρό κάνουμε μια αντιγραφή παραπάνω: "`ptrToNode->lnData = pnln->lnData`". Αν τα αντικείμενα του *K* είναι μεγάλα αυτή η αντιγραφή κοστίζει.

Το δεύτερο πρόβλημα μπορείς να το λύσεις –με άλλον αλγόριθμο– αλλά το πρώτο δεν λύνεται. Η διαγραφή κόμβου *ακυρώνει* τον προσεγγιστή. Αυτό είναι ένα γενικότερο πρόβλημα με τις περιέχουσες κλάσεις: *οι διαγραφές (αλλά και οι εισαγωγές) τιμών μπορεί να ακυρώνουν κάποιους προσεγγιστές*.

25.7.4 Μια Άλλη Στοίβα

Υποσχεθήκαμε ότι «*Στη συνέχεια θα δούμε μια πράγματι δυναμική στοίβα πολύ πιο χρήσιμη.*» Θα την υλοποιήσουμε χρησιμοποιώντας –όχι πίνακα– αλλά λίστα με απλή σύνδεση.

Στην προηγούμενη υποπαράγραφο είδαμε ένα «άδειο» σχέδιο του περιγράμματος (*StackT*) που θα γράψουμε. Το μοναδικό κρυμμένο μέλος είναι:

```
   SListT<K> cont;
```

Ο ερήμην δημιουργός είναι απλούστατος:

```
   StackT() { };
```

αφού μόνη η δήλωση της λίστας δημιουργεί μια κενή στοίβα.

Ο δημιουργός αντιγραφής

```
   StackT( const StackT& aSt );
```

ορίζεται ως:

```
template < class K, size_type sz >
StackT<K,sz>::StackT( const StackT<K,sz>& other )
   :  cont( other.cont )  {  }
```

και έχει λυμένα τα προβλήματά του από τον δημιουργό αντιγραφής της λίστας.

Ο (σωστός) τελεστής εκχώρησης της λίστας μας επιτρέπει να μην γράψουμε τελεστή εκχώρησης για τη στοίβα.

Η *push()* ορίζεται ως εξής:

```
template < class K, size_type sz >
void StackT<K,sz>::push( const K& v )
{
   try {  cont.push_front( v );  }
   catch( SListTXptn& x )
   {  throw StackTXptn( "push", StackTXptn::stackFull );  }
} // StackT<K,sz>::push
```

αλλά εδώ έχουμε ένα πρόβλημα: Η *SListT*::*push_front()* είναι δηλωμένη "`private`"· θα πρέπει να την κάνουμε "`public`".

Για να ορίσουμε την

```
template < class K, size_type sz >
void StackT<K,sz>::pop()
{
   if ( cont.empty() )
      throw StackTXptn( "pop", StackTXptn::stackEmpty );
   cont.pop_front();
} // StackT<K,sz>::pop
```

θα πρέπει να ορίσουμε την *SListT*::*pop_front()*:

```
template < typename K >
void SListT<K>::pop_front()
{
   if ( slHead == slTail )
      throw SListTXptn( "pop_front", SListTXptn::listEmpty );
   ListNode* pnln( slHead );
   slHead = slHead->lnNext;
   delete pnln;
} // SListT<K>::pop_front
```

Τώρα, η *getTop()* ορίζεται ως εξής:

```
template < class K, size_type sz >
const K& StackT<K,sz>::getTop() const
{
   if ( cont.empty() )
      throw StackTXptn( "getTop", StackTXptn::stackEmpty );
   return *cont.begin();
} // StackT<K,sz>::getTop
```

Τέλος, τα δύο κατηγορήματα ορίζονται `inline` ως εξής:

```
   bool isEmpty() const {  return cont.empty();  }
   bool isFull() const {  return false;  }
```

Το δεύτερο είναι λίγο περίεργο; Όχι και τόσο αφού για να γεμίσει η στοίβα θα πρέπει να πάρουμε όλη τη διαθέσιμη μνήμη του υπολογιστή μας. Και τότε, αν αποπειραθούμε να καλέσουμε την *push()* θα πάρουμε εξαίρεση με κωδικό *StackTXptn*`::`*stackFull* που θα προκληθεί από τη σύλληψη εξαίρεσης με κωδικό *SListTXptn*`::`*allocFailed* που θα ρίξει η *push_front()*.

Δηλαδή όλες οι αλλαγές είναι στην υλοποίηση και το τμήμα διεπαφής δεν αλλάζει; Ακριβώς! Και δεν θα πρέπει να ξεφορτωθούμε τη δεύτερη παράμετρο ("`size_type sz = 100`") καθώς και την *isFull()*; Μπορούμε να το κάνουμε και έτσι· σε «πραγματικές» εφαρμογές όμως θα πρέπει να σκεφτείς το εξής: αν θέλουμε να περάσουμε στην καινούρια υλοποίηση, θα αλλάξουμε όλα τα προγράμματα που έχουμε γράψει και χρησιμοποιούν το περίγραμμα της στοίβας; Η απάντηση στο ερώτημα αυτό είναι συνήθως «όχι». Βέβαια, σε κάποιο πρόγραμμα που καλεί την *isFull()* μπορεί να συμβεί το εξής: οι

```
   if ( !aStack.isFull() )
      aStack.push( v );
```

μπορεί να προκαλέσουν έγερση εξαίρεσης *StackTXptn* με κωδικό *stackFull*! Ε, συμβαίνουν αυτά όταν κάνουμε βελτιώσεις…

Αλλαγές όμως θα έχουμε στο περίγραμμα *SListT*:

- Η *push_front()* έγινε "`public`" και
- Γράψαμε άλλη μια μέθοδο, την *pop_front()* που είναι και αυτή "`public`".

Αυτές μάλλον δεν προκαλούν προβλήματα σε εφαρμογές που χρησιμοποιούν το περίγραμμα.[11]

Παρατήρηση:▶

Για το πρόγραμμα του παραδείγματος της §25.7 η υλοποίηση με τον πίνακα είναι μάλλον πιο κατάλληλη. Για να το καταλάβεις υπολόγισε τη (σπατάλη σε) μνήμη αν υλοποιήσουμε τις δύο στοίβες με λίστα.◀

25.8 Έξυπνα Βέλη

Στην §16.7 εντοπίσαμε ορισμένα προβλήματα που μπορεί να έχουμε όταν διαχειριζόμαστε δυναμική μνήμη με μεταβλητές-βέλη.

[11] Αν κάποιος έγραψε κλάση ή περίγραμμα που να κληρονομεί το *SListT*… Εδώ μπορεί να υπάρχει πρόβλημα.

- **Διαρροή μνήμης** (memory leakage): «*το πρόγραμμά μας έχει δυναμική(-ές) μεταβλητή(-ές) αλλά δεν έχει τρόπο –δηλαδή κάποιο βέλος– για να τη χειριστεί (ούτε να την ανακυκλώσει).*»
- **Μετέωρο βέλος** (pending ή dangling pointer): αν δύο ή περισσότερα βέλη δείχνουν την ίδια δυναμική μεταβλητή και την ανακυκλώσουμε τότε κάποιο(-α) βέλος(-η) μένει(-ουν) *μετέωρο*(-α) με την έννοια ότι δείχνει θέση της μνήμης που δεν ελέγχεται από το πρόγραμμά μας.
- **Πολλαπλή διαγραφή (ανακύκλωση):** Αν το βέλος *p* δείχνει κάποια περιοχή της δυναμικής μνήμης οι εντολές "`delete p; delete p;`" μπορεί να προκαλέσουν καταστροφικό λάθος στο σύστημα διαχείρισης της δυναμικής μνήμης.

Τα **έξυπνα βέλη** (smart pointers) είναι προγραμματιστικά εργαλεία που χρησιμοποιούμε για να ξεπεράσουμε τα προβλήματα που απαριθμήσαμε παραπάνω.

Παράδειγμα έξυπνου βέλους μας δίνει η STL της C++. Αναφερόμαστε στο περίγραμμα κλάσης *auto_ptr*. Αυτό το περίγραμμα κλάσης θα μελετήσουμε εδώ χωρίς να δεσμευόμαστε από κάποια συγκεκριμένη υλοποίηση. Για να μην υπάρχουν μπερδέματα, θα ονομάσουμε το δικό μας περίγραμμα *AutoPtr*.[12]

Η ιδέα είναι να κρύψουμε το βέλος μέσα σε ένα αντικείμενο που θα το κάνουμε να συμπεριφέρεται σαν βέλος: Μετά τις

```
Date*            pd( new Date(2010, 11, 26) );
AutoPtr< Date >  apd( pd );
```

το "`*apd`" και το "`*pd`" θα πρέπει να είναι το ίδιο πράγμα. Το ίδιο ισχύει και για τα "`apd->getYear()`", "`apd->getMonth()`", "`apd->getDay()`" και τα "`pd->getYear()`", "`pd->getMonth()`", "`pd->getDay()`" αντιστοίχως.

Κατά τα άλλα θα ρυθμίσουμε τη συμπεριφορά του *AutoPtr*`<`*K*`>` για να αντιμετωπίσουμε τα προβλήματα που παραθέσαμε παραπάνω.

Αυτό το κάναμε ήδη μια φορά με τους προσεγγιστές. Εκεί είδαμε ότι για να έχει το αντικείμενο συμπεριφορά βέλους θα πρέπει η (παραμετρική) κλάση μας να είναι:

```
template < typename K >
class Iterator
{
public:
   explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
   K& operator*() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListTXptn( "Iterator::operator*",
                           SListTXptn::listEnd );
      return ( iPNode->lnData );
   } // operator*
   K* operator->() const
   {
      if ( iPNode->lnNext == 0 )
         throw SListXptn( "Iterator::operator->",
                          SListXptn::listEnd );
      return ( &(iPNode->lnData) );
   } // SList::Iterator::operator->
private:
   ListNode* iPNode;
   ListNode* getPNode() const {  return iPNode;  }
}; // Iterator
```

Ξεκινούμε και εδώ το περίγραμμα κλάσης ως εξής:

---

[12] Στο (Alexandrescu 2001) εξετάζονται άλλες επιλογές που υπάρχουν για τη σχεδίαση μιας παραμετρικής κλάσης για έξυπνα βέλη. Μπορείς να βρεις το Κεφ. 7 (για τα έξυπνα βέλη) αυτού του βιβλίου στο `http://ptgmedia.pearsoncmg.com/images/9780201704310/samplepages/0201704315.pdf`.

```
template < typename K >
class AutoPtr
{
// . . .
private:
   K* ap;
}; // AutoPtr
```

Η διαφορά μας από τον προσεγγιστή είναι ότι εκεί το βέλος *iPNode* στόχευε έναν κόμβο μέσα στον οποίο υπάρχει το αντικείμενο που μας ενδιαφέρει ενώ τώρα στοχεύει το ίδιο το αντικείμενο.

Ο δημιουργός είναι ολόϊδιος:

```
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
```

Αλλά τώρα θα χρειστούμε –η πρώτη σημαντική διαφορά– και *καταστροφέα που θα ανακυκλώνει τον στόχο*:

```
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

Για την *getPNode()* υπάρχουν δυο διαφορές (με βάση αυτά που ισχύουν για το *auto_ptr*):

- Η πιο σημαντική διαφορά: είναι "`public`".
- Η λιγότερο σημαντική: θα της αλλάξουμε το όνομα σε "*get*".

```
   K* get() const {  return ap;  }
```

Στην επιφόρτωση του "`operator*`" και του "`operator->`" έχουμε πρόβλημα: Όλες οι μέθοδοι της *auto_ptr* έχουν προδιαγραφή εξαιρέσεων "`throw()`". Το C++03 δίνει τις εξής προδιαγραφές για την `operator*()` του *auto_ptr* που ακολουθούμε:

`X& operator*() const throw();`

Προϋπόθεση: `get() != 0`

Επιστρέφει: `*get()`

Για την `operator->()` μας λέει:

`X* operator->() const throw();`

Επιστρέφει: `get()`

Οι συναρτήσεις που γράψαμε για τον προσεγγιστή μπορεί να ρίξουν εξαίρεση! Εδώ τι θα κάνουμε; Θα ακολουθήσουμε το πρότυπο:

```
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
```

Έτσι, αν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα αντικείμενο *AutoPtr* που έχει *ap* `==` 0 θα έχεις –όταν εκτελείται η `operator*()`– το πρόβλημα που έχεις όταν προσπαθήσεις να κάνεις αποπαραπομπή σε ένα σύνηθες μηδενικό βέλος. Στην περίπτωση του "`->`" η εκτέλεση της `operator->()` γίνεται κανονικώς και το πρόβλημα θα παρουσιαστεί όταν θα δράσει ο "`->`" στο αποτέλεσμα της συνάρτησης.

Ας πούμε ότι έχουμε δηλώσει:

```
   AutoPtr<string> aps( new string("abc") );
```

Οι εντολές

```
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;
```

θα δώσουν:

```
abc
qwerty
```

Μέχρι εδώ έχουμε:

```
template < typename K >
class AutoPtr
{
   explicit AutoPtr( K* p = 0 )  {  ap = p;  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
```

```
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
private:
   K* ap;
}; // AutoPtr
```

Τι έχουμε κερδίσει με αυτά; Αν έχουμε:

```
void f( int v )
{
   int* p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
   delete p;
} // f
```

παρ' όλο που φροντίσαμε να βάλουμε τη "`delete p`", αν ριχτεί εξαίρεση θα έχουμε διαρροή μνήμης. Αν όμως βάλουμε:

```
void f( int v )
{
   AutoPtr<int> p( new int(v) );
// . . .
   if ( . . . ) throw SomeXptn( . . . );
// . . .
   int q( 25 + *p );
// . . .
} // f
```

τα πράγματα αλλάζουν: Είτε η εκτέλεση της συνάρτησης τελειώσει κανονικώς είτε διακοπεί λόγω ρίψης εξαίρεσης η καταστροφή της *p* ανακυκλώνει και τη μνήμη που κατέχει η *p*. Όπως καταλαβαίνεις, δεν χρειαζόμαστε πια τον `delete`, αφού έχουμε αυτόματη ανακύκλωση.

Θα δούμε τώρα δύο χαρακτηριστικές μεθόδους της *AutoPtr*. Η πρώτη είναι η *release()*. Το πρότυπο μας λέει για τη *release()* της *auto_ptr*:

```
X* release() throw();
```

Επιστρέφει: `get()`

Απαίτηση: το αντικείμενο (`*this`) έχει το μηδενικό βέλος (`NULL`).

Δηλαδή: επιστρέφει την τιμή του «κρυμμένου» βέλους αλλά επιπλέον το μηδενίζει.

```
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
```

Ας δούμε τη διαφορά της από τη *get()* με ένα παράδειγμα. Μετά τη δήλωση

```
   auto_ptr< Date >  apd( new Date(2010, 11, 26) );
```

οι εντολές:

```
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
```

δίνουν:

```
26.11.2010
26.11.2010
26.11.2010
it is NULL
```

Η τελευταία γραμμή δείχνει ότι, μετά την *apd.release()*, το βέλος που κρύβεται μέσα στο *apd* μηδενίζεται. Βέβαια η τιμή του πέρασε στο *pd1*, όπως φαίνεται από την προτελευταία

γραμμή. Με τη "`Date* pd0(apd.get())`" αντιγράφεται η τιμή του βέλους του *apd* στο *pd0* χωρίς να αλλάζει, όπως δείχνουν οι δύο πρώτες γραμμές.

Η δεύτερη μέθοδος είναι η *reset()*:

```
void reset( X* p=0 ) throw();
```

Ενέργεια: Αν `get() != p` τότε `delete get()`.
Απαίτηση: το αντικείμενο (`*this`) έχει το βέλος *p*.
Με αυτήν αλλάζουμε το κρυμμένο βέλος. Αλλά προσοχή: *αν το παλιό βέλος έχει την κυριότητα κάποιου αντικειμένου αυτό ανακυκλώνεται πριν από την αλλαγή της τιμής του βέλους.*

```
void reset( K* p=0 )
{  if ( ap != p ) {  delete ap;  ap = p;  }  }
```

Πρόσεξε ότι αν δώσουμε:

```
api1.reset();
```

ανακυκλώνεται το αντικείμενο που κατέχει το *api1*. Δηλαδή, μπορούμε έτσι να ανακυκλώνουμε τη μνήμη όποτε θέλουμε.

Ας δούμε τώρα τον «δημιουργό αντιγραφής» και τον τελεστή εκχώρησης.

Το C++03 μας λέει:

```
auto_ptr( auto_ptr& a ) throw();
```

Ενέργεια: Εκτελείται η κλήση "`a.release()`".
Απαίτηση: Το `*this` κρατάει το βέλος που επιστρέφει η `a.release()`.

Για το *AutoPtr* γράφουμε:

```
AutoPtr( AutoPtr& other ) {  ap = other.release();  }
```

που ισοδυναμεί με:

```
AutoPtr( AutoPtr& other ) {  ap = other.ap;  other.ap = 0;  }
```

Τι δημιουργός αντιγραφής είναι αυτός; Η μοναδική εντολή φέρνει στο *ap* την τιμή του *other.ap* αλλά, επί πλέον, μηδενίζει το *other.ap*· για να γίνεται αυτό *δεν βάλαμε* "`const`" στην παράμετρο. Στην πραγματικότητα εδώ έχουμε έναν **δημιουργό μεταβίβασης** (move constructor).[13]

Για τον τελεστή εκχώρησης οι προδιαγραφές είναι:

```
auto_ptr& operator=( auto_ptr& a ) throw();
```

Προϋπόθεση: Μπορεί να εκτελεσθεί η "`delete get()`".
Ενέργεια: `reset( a.release() )`.
Επιστρέφει: `*this`.

Για το *AutoPtr* μπορούμε να γράψουμε:

```
AutoPtr& operator=( AutoPtr& rhs )
{  reset( rhs.release() );  return *this;  }
```

που είναι ισοδύναμο με:

```
AutoPtr& operator=( AutoPtr& rhs )
{  if ( ap != rhs.ap )
   {  delete ap;  ap = rhs.ap;  rhs.ap = 0;  }
   return *this;  }
```

Τώρα μπορείς να καταλάβεις γιατί πολλές φορές γράφαμε για «*αντιγραφικό τελεστή εκχώρησης*»· όπως βλέπεις εδώ έχουμε έναν *μεταβιβαστικό* τελεστή εκχώρησης. Αφού η *reset()* βάζει ως νέο βέλος αυτό που επιστρέφει η `rhs.release()` το βέλος του *rhs* δεν αντιγράφεται αλλά μηδενίζεται. Για να είναι αυτό δυνατό η παράμετρος δεν έχει "`const`".

Όπως βλέπεις, όλες οι μέθοδοι έχουν τα χαρακτηριστικά που πρέπει για να γλυτώσουμε από τα προβλήματα που παραθέσαμε στην αρχή:

- Δεν επιτρέπονται αντίγραφα βελών.

---

[13] Στο C++11 ο δημιουργός μεταβίβασης ορίζεται με πιο συγκεκριμένο τρόπο και έχει την ίδια θέση με τον συνήθη δημιουργό αντιγραφής.

- Όταν αλλάζουμε την τιμή βέλους που έχει την κυριότητα κάποιου δυναμικού αντικειμενου ανακυκλώνεται και το αντικείμενο.
- Κάθε φορά που ανακυκλώνεται το δυναμικό αντικείμενο μηδενίζεται η τιμή του βέλους που το είχε.

Υπάρχει όμως μια εξαίρεση: η *get()*!

Να λοιπόν πώς θα είναι το **AutoPtr.h**:

```
#ifndef _AUTOPTR_H
#define _AUTOPTR_H

template < typename K >
class AutoPtr
{
public:
   explicit AutoPtr( K* p=0 )  {  ap = p;  }
   AutoPtr( AutoPtr& other ) {  ap = other.release();  }
   ~AutoPtr()  {  delete ap;  ap = 0;  }
   AutoPtr& operator=( AutoPtr& rhs )
   {  reset( rhs.release() );  return *this;  }
   K* get() const {  return ap;  }
   K& operator*() const {  return *ap;  }
   K* operator->() const {  return ap;  }
   K* release() {  K* fv( ap );  ap = 0;  return fv;  }
   void reset( K* p=0 )
   {  if ( ap != p ) {  delete ap;  ap = p;  }  }
private:
   K* ap;
}; // AutoPtr

#endif // _AUTOPTR_H
```

Το παρακάτω πρόγραμμα περιέχει όλες τις δοκιμές που είδαμε παραπάνω:

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

#include "AutoPtr.h"
#include "Date.cpp"

using namespace std;

int main()
{
   AutoPtr< string > aps( new string("abc") );
   cout << *aps << endl;
   aps->assign( "qwerty" );
   cout << *aps << endl;

   AutoPtr< Date >  apd( new Date(2010, 11, 26) );
   cout << *apd << endl;
   Date* pd0( apd.get() );
   cout << *apd << endl;

   Date* pd1( apd.release() );
   cout << *pd1 << endl;
   if ( apd.get() == 0 )  cout << "it is NULL" << endl;
}
```

Πρόσεξε το εξής: Ενώ έχουμε δύο "**new**" δεν υπάρχουν "**delete**" διότι δεν χρειάζονται· την ανακύκλωση των δυναμικών αντικειμένων τη φροντίζουν οι καταστροφείς. Είδαμε λοιπόν έναν τρόπο για να εφοδιάσουμε τη C++ με μηχανισμό «**αποκομιδής απορριμάτων**» (garbage collection). Βέβαια δεν είναι πλήρης, αφού δεν καλύπτει και δυναμικούς πίνακες αλλά είναι ένα πολύ καλό εργαλείο.

25.8.1 *std*: : *auto_ptr*

Αν στο παραπάνω πρόγραμμα με τα παραδείγματα αλλάξεις:

- το "`#include "AutoPtr.h"`" σε "`#include <memory>`" και
- το "`AutoPtr`" σε "`auto_ptr`"

θα πάρεις τα ίδια αποτελέσματα· αυτήν τη φορά όμως με χρήση του περιγράμματος (*std*: :)*auto_ptr* που μας δίνει η C++.

Δες την υλοποίηση του `auto_ptr` στο `memory` της C++ που χρησιμοποιείς και πιθανότατα θα δεις ότι είναι σαν αυτήν του `AutoPtr` που δώσαμε πιο πάνω. Τώρα που είδες πώς και γιατί γράφηκε αυτή η παραμετρική κλάση ας δούμε πώς μπορείς να τη χρησιμοποιείς.

1. Μπορείς να δηλώσεις μεταβλητές ως εξής:

```
auto_ptr<int> api1( new int(375) );
auto_ptr<string> aps1( new string("F.C.Barcelona") );
auto_ptr<int> api2;
auto_ptr<string> aps2( aps1 );
```

- Η πρώτη δήλωση έχει ως αποτέλεσμα: η *api1* να έχει κυριότητα μιας δυναμικής μεταβλητής (τύπου `int`) με έχει τιμή 375.
- Η δεύτερη δήλωση έχει αποτέλεσμα: η *aps1* κατέχει μια δυναμική μεταβλητή (τύπου *string*) που έχει τιμή `"F.C.Barcelona"`.
- Το αποτέλεσμα της τρίτης δήλωσης: η *api2* να δεν κατέχει κάποια δυναμική μεταβλητή.
- Η τέταρτη δήλωση έχει ως αποτέλεσμα η *aps2* να πάρει την κυριότητα της δυναμικής μεταβλητής (τύπου `std::string`) που έχει τιμή `"F.C.Barcelona"` και ήταν αρχικώς στην κατοχή της *aps1*. Η *aps1* δεν κατέχει πια κάποια δυναμική μεταβλητή.

Προσοχή όμως!

α) *Τα παρακάτω απαγορεύονται*:

```
int  k( 255 );
auto_ptr<int> api0( &k );
```

Δεν μπορείς να δώσεις στην κατοχή ενός έξυπνου βέλους μια μεταβλητή που δεν είναι δυναμική. Γιατί; Διότι η ώρα του "`delete`" θα έλθει οπωσδήποτε! Δυστυχώς, ο μεταγλωττιστής δεν θα σε αποτρέψει! Όλα θα είναι στη δική σου την ευθύνη!

β) Κατ' αρχήν μπορείς να συνεχίσεις να χειρίζεσαι μια δυναμική μεταβλητή με «παραδοσιακό» βέλος και μετά την κατοχή της από ένα έξυπνο βέλος. Π.χ. οι παρακάτω:

```
int* pi1( new int(511) );
auto_ptr<int> api( pi1 );
cout << *api << endl;
*pi1 = 375;
cout << *api << endl;
int* pi2( api.get() );
*pi2 = 101;
cout << *api << endl;
```

δίνουν:

```
511
375
101
```

Πάντως *κάτι τέτοιο είναι εξαιρετικά επικίνδυνο*!

2. Μπορείς να ελέγξεις την τιμή του κρυμμένου βέλους παίρνοντάς το με τη μέθοδο *get()*. Μπορείς να κάνεις αποπαραπομπή με το "`*`" όπως και με τα απλά βέλη.

Μετά τις παραπάνω δηλώσεις οι παρακάτω εντολές

```
if ( api1.get() != 0 ) cout << *api1 << endl;
                       else cout << " api1 NULL" << endl;
if ( aps1.get() != 0 ) cout << *aps1 << endl;
                       else cout << " aps1 NULL" << endl;
if ( api2.get() != 0 ) cout << *api2 << endl;
                       else cout << " api2 NULL" << endl;
```

```
   if ( aps2.get() != 0 ) cout << *aps2 << endl;
                     else cout << " aps2 NULL" << endl;
```

θα δώσουν:

```
375
 aps1 NULL
 api2 NULL
F.C.Barcelona
```

3. Να ξαναγυρίσουμε στις ιδιαιτερότητες του «δημιουργού αντιγραφής» που, όπως είπαμε, είναι *δημιουργός μεταβίβασης*. Μετά την τέταρτη περίπτωση του σημείου 1, δες και αυτό:

```
   cout << *api1 << endl;
   cout << *(f(api1)) << endl;
   if ( api1.get() != 0 ) cout << *api1 << endl;
                     else cout << " api1 NULL" << endl;
```

όπου:

```
auto_ptr<int> f( auto_ptr<int> x )
{
   *x += 1;
   return x;
} // f
```

Αποτέλεσμα:

```
375
376
 api1 NULL
```

Αρχικώς η *api1* κατέχει βέλος προς δυναμική μεταβλητή που έχει τιμή 375. Όταν καλείται η *f()*, η *api1* περνάει ως παράμετρος τιμής. Αυτό σημαίνει ότι χρησιμοποιείται ο δημιουργός αντιγραφής με αποτέλεσμα η *api1* να χάνει την κατοχή του βέλους προς τη δυναμική μεταβλητή. Την κατοχή έχει πια η τοπική μεταβλητή της συνάρτησης *x*.

Με την εκτέλεση της `return` και η *x* χάνει την κατοχή του βέλους, αφού και πάλι εκτελείται ο δημιουργός αντιγραφής, αλλά αυτό είναι κάτι που δεν μπορούμε να το δούμε ούτε και μας ενδιαφέρει!

4. Μπορείς να αλλάξεις την τιμή μιας τέτοιας μεταβλητής είτε με τη *reset()* είτε με τον (μεταβιβαστικό) τελεστή εκχώρησης "`=`".

Ας θυμίσουμε την επικεφαλίδα της *reset()*:

```
   void reset( K* p = 0 )
```

Δηλαδή, η *reset()* περιμένει όρισμα που είναι σύνηθες βέλος. Έτσι, οι εντολές:

```
   api1.reset( new int(1023) );
   cout << *api1 << endl;
   cout << *aps2 << endl;
   aps2.reset( new string("Kavala F.C.") );
   cout << *aps2 << endl;
```

θα δώσουν:

```
1023
F.C.Barcelona
Kavala F.C.
```

Όπως στη δήλωση, έτσι και στη *reset()*,

α) *δεν επιτρέπεται να περάσεις βέλος προς μια μη δυναμική μεταβλητή*,

β) παρ' όλο που μπορείς να χειρίζεσαι μια δυναμική μεταβλητή και με το απλό βέλος, *από τη στιγμή που το πέρασες σε ένα έξυπνο βέλος μην το ξαναπειράξεις* (το απλό βέλος).

Ο τελεστής "`=`" δουλεύει ως εξής:

```
   auto_ptr& operator=( auto_ptr& rhs ) throw()
   {  reset( rhs.release() );  return *this;  }
```

Πρόσεξε ότι η παράμετρος είναι αναφοράς. Επομένως, μπορείς να μεταβιβάσεις την κατοχή βέλους από άλλη μεταβλητή-έξυπνο-βέλος. Μπορείς λοιπόν να βάλεις:

```
double* pd2( new double(7.5) );
auto_ptr<double> apd2( pd2 );
auto_ptr<double> apd3;
apd3 = apd2;
```

αλλά όχι:

```
apd3 = auto_ptr<double>( new double(7.5) );
```

ούτε:

```
apd3 = f( pd2 );
```

Φυσικά, το σπουδαιότερο που πρέπει να θυμάσαι είναι ότι με *τον τελεστή αυτόν γίνεται μεταβίβαση τιμής και όχι εκχώρηση όπως την ξέρουμε*. Με την:

```
apd3 = apd2;
```

η *apd3* γίνεται κάτοχος του βέλους που είχε η *apd2* αλλά η *apd2* χάνει την κατοχή και καθαρίζεται (*apd2*.*ap* = 0).

5. Με τη *reset()* μπορείς να ανακυκλώσεις τη δυναμική μνήμη που διαχειρίζεται ένα έξυπνο βέλος (περίπου ό,τι κάνεις με τον "`delete`" για τα συνήθη βέλη). Π.χ.:

```
apd3.reset();
```

6. Με τη *release()* μπορείς να ξαναπάρεις τον έλεγχο από το έξυπνο βέλος:

```
double* pd( apd3.release() );
```

25.8.2 Συνελόντι Ειπείν...

Τα έξυπνα βέλη είναι πολύ καλά εργαλεία για να γράφεις πιο σίγουρα προγράμματα με λιγότερα λάθη. Η πιο απλή περίπτωση είναι τα βέλη που παράγονται από το περίγραμμα `auto_ptr`, που βρίσκεται στην πάγια βιβλιοθήκη της C++.

Αν προσπαθήσεις να τα μάθεις και να τα χρησιμοποιείς, τα σημεία που πρέπει να προσέξεις είναι τα εξής:

- Ο δημιουργός αντιγραφής και ο "=" κάνουν μεταβίβαση τιμής και όχι συνήθη αντιγραφή.
- Ο "`delete`" συνήθως δεν χρειάζεται αλλά όπου είναι απαραίτητος αντικαθίσταται από κλήση της "`reset()`".

Μια σοβαρή έλλειψη του *auto_ptr* είναι ότι δεν μπορεί να χειρισθεί δυναμικούς πίνακες.

Πάντως, στο C++11 αποθαρρύνεται η χρήση του *auto_ptr* (που μπορεί να μην υπάρχει σε επόμενες τυποποιήσεις) αφού δίνεται ένα νέο περίγραμμα κλάσης για έξυπνα βέλη, το *unique_ptr*, με πολύ περισσότερες δυνατότητες.

Το C++11 δίνει ακόμη δύο περιγράμματα κλάσεων για έξυπνα βέλη, τα *shared_ptr* και *weak_ptr*, που είναι περίπου αυτά που υπάρχουν στη βιβλιοθήκη Boost με τα ίδια ονόματα.[14]

25.9 Ένα Χρήσιμο Περίγραμμα Κλάσης

Βάζοντας στο πρόγραμμά σου "`#include <limits>`" μπορείς να χρησιμοποιήσεις το περίγραμμα κλάσης *std*::*numeric_limits*. Η δήλωσή του είναι:

```
template< typename T >
class numeric_limits
{
public:
   static const bool is_specialized = false;
   static T min() throw();
   static T max() throw();
   static const int digits = 0;
   static const int digits10 = 0;
```

[14] `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`

```
   static const bool is_signed = false;
   static const bool is_integer = false;
   static const bool is_exact = false;
   static const int radix = 0;
   static T epsilon() throw();
   static T round_error() throw();
   static const int min_exponent = 0;
   static const int min_exponent10 = 0;
   static const int max_exponent = 0;
   static const int max_exponent10 = 0;
   static const bool has_infinity = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const float_denorm_style has_denorm = denorm_absent;
   static const bool has_denorm_loss = false;
   static T infinity() throw();
   static T quiet_NaN() throw();
   static T signaling_NaN() throw();
   static T denorm_min() throw();
   static const bool is_iec559 = false;
   static const bool is_bounded = false;
   static const bool is_modulo = false;
   static const bool traps = false;
   static const bool tinyness_before = false;
   static const float_round_style round_style = round_toward_zero;
};
```

Μα τι κλάσεις θα βγάλουμε από εδώ; *Τα πάντα είναι "`static`"!* Θα το καταλάβεις από τα παραδείγματα που θα δεις στη συνέχεια.

Τα ονόματα των μελών περιγράφουν αρκετά καλά το νόημά τους. Για μερικά από αυτά μπορεί να πρέπει να γυρίσεις στο Κεφ. 17. Στο `limits` υπάρχουν και εξειδικεύσεις για τους βασικούς τύπους `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double` και `long double`.

Ορισμένα μέλη –όπως *min_exponent*, *min_exponent10*, *max_exponent*, *max_exponent10* κλπ και οι συναρτήσεις *epsilon()*, *round_error()* κλπ– έχουν νόημα μόνο για τύπους κινητής υποδιαστολής, δηλαδή τους `float`, `double` και `long double` (και άλλους που τυχόν δίνει κάποια υλοποίηση). Αν κάποιο μέλος δεν έχει νόημα για κάποιον τύπο τότε στην εξειδίκευση αφήνουμε τιμή "`0`" ή "`false`"όπως φαίνεται στην παραπάνω δήλωση.

Με χρήση των κατάλληλων εξειδικεύσεων μπορούμε να ξαναγράψουμε το προγραμματάκι (§2.5):

```
#include <iostream>
#include <climits>
using namespace std;
int main()
{
   cout << "INT_MIN = " << INT_MIN << "    INT_MAX = "
        << INT_MAX << endl;
}
```

ως εξής:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   cout << "INT_MIN = " << numeric_limits<int>::min()
        << "    INT_MAX = " << numeric_limits<int>::max() << endl;
}
```

Δηλαδή:

- Αντί για "`#include <climits>`" βάλαμε "`#include <limits>`".
- Αντί για "`INT_MIN`" βάλαμε "`numeric_limits<int>::min()`".

- Αντί για "`INT_MAX`" βάλαμε "`numeric_limits<int>::max()`".

Ο πρώτος τρόπος γραφής είναι η κληρονομιά από τη C. Ο δεύτερος τρόπος, με το πολύ γράψιμο, είναι «καθαρή C++».

Παρομοίως, το πρόγραμμα:

```
#include <iostream>
#include <cfloat>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << DBL_MIN
        << "    DBL_MAX = " << DBL_MAX << endl;
   cout << "DBL_EPSILON = " << DBL_EPSILON << endl;
}
```

θα γίνει:

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   cout << "DBL_MIN = " << numeric_limits<double>::min()
        << "    DBL_MAX = " << numeric_limits<double>::max()
        << endl;
   cout << "DBL_EPSILON = " << numeric_limits<double>::epsilon()
        << endl;
}
```

Από εδώ και πέρα θα προτιμούμε να γράφουμε πιο πολλά και να είμαστε «πιο C++» εκτός από την εξής περίπτωση: όταν ελέγχουμε τα αποτελέσματα συναρτήσεων της C, όπως αυτά που είδαμε στην §24.1.3. Εκεί θα πρέπει να βάζουμε τις προκαθορισμένες σταθερές της C.

Οι κλάσεις που προκύπτουν από τις εξειδικεύσεις της *numeric_limits* είναι παραδείγματα **κλάσεων χαρακτηριστικών** (traits) και η εισαγωγή τους δεν έγινε μόνο για να γράφουμε προγράμματα «πιο C++». Θα δούμε τη χρησιμότητά τους με ένα

Παράδειγμα

Ας πούμε ότι θέλουμε να γράψουμε ένα

```
template < typename T >
class MyClass
{
   void f( ... )
   { . . . }
}
```

που θα μας δίνει στιγμιότυπα για ακέραιους τύπους.

Στη μέθοδο *f* θέλουμε τη μέγιστη τιμή του τύπου *T*. Με αυτά που ξέρουμε μέχρι τώρα θα πρέπει να γράψουμε μια πολλαπλή `if`:

```
if ( T είναι ο int )  {  T mt( INT_MAX ); ... }
else if ( T είναι ο long )  {  T mt( LONG_MAX ); ... }
else . . .
```

Οι συνθήκες "`T είναι ο int`" και οι άλλες παρόμοιες μεταφράζονται σε C++ με τη βοήθεια του `typeinfo`.

Χρησιμοποιώντας το περίγραμμα που είδαμε πιο πάνω τα πράγματα απλουστεύονται πάρα πολύ:

```
T mt( numeric_limits<T>::max() ); ...
```

Όπως καταλαβαίνεις, οι κλάσεις χαρακτηριστικών είναι πολύτιμο εργαλείο όταν έχουμε να γράψουμε περιγράμματα.

Παρατήρηση:▶

Αξίζει να πούμε κάτι ακόμη, σχετικώς με το πρώτο μας παράδειγμα, το περίγραμμα *BStringT*: Η C++ μας δίνει το περίγραμμα:

```
template< typename charT > struct char_traits
```

και δύο εξειδικεύσεις του:

```
template<> struct char_traits< char >;
template<> struct char_traits< wchar_t >;
```

Με τη χρήση τους θα μπορούσαμε να γράψουμε το *BStringT* πιο εύκολα (και να το κάνουμε πιο αποδοτικό.)◀

25.10 Ανακεφαλαίωση

Με ένα περίγραμμα κλάσης μπορείς να υλοποιήσεις μια δομή δεδομένων με όλες τις λειτουργίες της που να έχει ως παράμετρο τον τύπο του *περιεχομένου*. Η υλοποίηση της δομής για συγκεκριμένο τύπο προκύπτει ως στιγμιότυπο του περιγράμματος.

Οι πιθανές ιδιαιτερότητες που παρουσιάζονται στα στιγμιότυπα για ορισμένους τύπους αντιμετωπίζονται με *εξειδικεύσεις* ή *μερικές εξειδικεύσεις*.

Ενδιαφέρον έχουν οι κληρονομιές περιγραμμάτων:

- Μια κλάση μπορεί να κληρονομεί στιγμιότυπο περιγράμματος κλάσης.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί κάποια κλάση ώστε να χρησιμοποιεί εργαλεία που έχει η κλάση.
- Ένα περίγραμμα κλάσης μπορεί να κληρονομεί άλλο περίγραμμα κλάσης.

Δύο περιγράμματα κλάσεων που «περιτυλίγουν» βέλη ώστε να εμπλουτίσουν ή/και να αλλάξουν τις ιδιότητές τους είναι: οι *προσεγγιστές* (που ήδη ξέραμε) και τα *έξυπνα βέλη* (μια σύνοψη στην §25.8.2).

Οι *κλάσεις χαρακτηριστικών* όπως η *numeric_limits* είναι πολύ χρήσιμες για να γράφεις περιγράμματα κλάσεων.

Ασκήσεις

25-1 «Μάζεψε» όλα τα περιγράμματα συναρτήσεων για διαχείριση ψηφιοπινάκων σε περίγραμμα κλάσης *Bitmap*.

κεφάλαιο

26

Βιβλιοθήκη Παγίων Περιγραμμάτων – Standard Template Library (STL)

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα κάνουμε μια σύντομη γνωριμία με την STL.
Προσδοκώμενα αποτελέσματα:
Θα μπορείς να χρησιμοποιείς στα προγράμματά σου έτοιμες και καλές λύσεις για πολλά προβλήματα και δεν θα χρειάζεται να γράφεις τα πάντα από την αρχή. Όπως θα κατάλαβεις, θα είναι σαν να γράφεις τα προγράμματά σου σε γλώσσα προγραμματισμού πολύ υψηλού επιπέδου.
Έννοιες κλειδιά:

- *περιέχουσα κλάση, περιέχον*
- *προσεγγιστής*
- *αλγόριθμοι*
- *συναρτησιακό αντικείμενο*
- *ακολουθίες*
- *συνειρμικά περιέχοντα*
- *επιλογή περιέχοντος*

Περιεχόμενα:

26.1 Περιέχοντα, Προσεγγιστές και Αλγόριθμοι ... 991
26.1.1 Περιέχουσες Κλάσεις ... 992
26.1.2 Περί Προσεγγιστών… ... 994
26.1.3 Αλγόριθμοι ... 996
26.1.3.1 Τρεις Συναρτήσεις για «τα Πάντα» ... 1000
26.1.4 Συναρτησιακά Αντικείμενα ... 1003
26.2 Ακολουθίες ... 1005
26.2.1 Το Περίγραμμα "*vector*" ... 1008
26.2.1.1 Εξαιρέσεις, Απόδοση και Άλλα ... 1011
26.2.1.2 Και μια Εξειδίκευση: "`vector<bool>`" ... 1012
26.2.2 Το Περίγραμμα "*deque*" ... 1013
26.2.3 Το Περίγραμμα "*list*" ... 1014
26.2.4 Ποια Ακολουθία να Διαλέξω; ... 1018
26.3 Συνειρμικά Περιέχοντα ... 1018
26.3.1 Το Περίγραμμα "*set*" ... 1020
26.3.1.1 Σχέσεις και Πράξεις Συνόλων ... 1023
26.3.2 Το Περίγραμμα "*map*" ... 1027
26.3.3 Τα Περιγράμματα "*multiset*" και "*multimap*" ... 1030
26.3.4 Διάταξη Στοιχείων ... 1031
26.4 Ποιο Περιέχον να Διαλέξω; ... 1033
26.5 Άλλα Περιγράμματα ... 1034
26.5.1 Το Περίγραμμα "*bitset*" ... 1034
26.5.2 Το Περίγραμμα "*complex*" ... 1037
26.6 Τι (Πρέπει να) Έμαθες στο Κεφάλαιο Αυτό ... 1038

Ασκήσεις 1038

Εισαγωγικές Παρατηρήσεις:
Η **Βιβλιοθήκη Παγίων Περιγραμμάτων** (Standard Template Library – STL) περιλαμβάνει τέσσερα είδη συνιστωσών:

- *περιέχουσες κλάσεις,*
- *προσεγγιστές,*
- *αλγόριθμους,*
- *συναρτησιακά αντικείμενα.*

Για την υλοποίησή τους χρησιμοποιούνται

- **παραχωρητές μνήμης** (allocators) και
- κλάσεις χαρακτηριστικών.

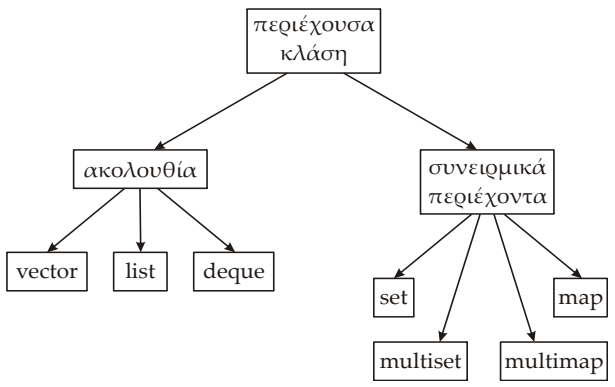
Στα παραδείγματα κλάσεων με δυναμικούς πίνακες βάλαμε έναν μηχανισμό που ήταν παντού ο ίδιος (με ή χωρίς τη *renew()*). Ο πειρασμός να γράψουμε ένα περίγραμμα κλάσης για δυναμικό μονοδιάστατο πίνακα με αυτορυθμιζόμενο μέγεθος είναι μεγάλος! Δεν χρειάζεται: η STL έχει τέτοιο εργαλείο.

Ας δούμε ένα απλό πρόβλημα και ας το λύσουμε με αυτά που ξέρουμε:

Θέλουμε ένα πρόγραμμα που θα διαβάζει από το πληκτρολόγιο άγνωστο πλήθος φυσικών, θα τους ταξινομεί και θα τους δείχνει στην οθόνη.

Χρησιμοποιώντας τη *renew()* από το `MyTmpltLib.h` και την *qsort()* που είδαμε στην §16.10 έχουμε:

```
#include <iostream>
#include <cstdlib>

#include "MyTmpltLib.h"

using namespace std;

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

int main()
{
   const unsigned int inc( 5 );
   size_type size( inc );
   unsigned int* array( new unsigned int[inc] );

   int x;
   size_type count( 0 );
   cin >> x;
   while ( x >= 0 )
   {
      if ( count >= size )
      {
         renew( array, count, size+inc );
         size += inc;
      }
      array[count] = x;
      ++count;
      cin >> x;
   } // while
   qsort( array, count, sizeof(int), compare );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```



Σχ. 26-1 Οι περιέχουσες κλάσεις της STL.

Στη συνέχεια θα δούμε διαφορετικές λύσεις που μπορούμε να γράψουμε με τα εργαλεία της STL.

26.1 Περιέχοντα, Προσεγγιστές και Αλγόριθμοι

Ένας άλλος τρόπος να λύσουμε το πρόβλημα που δώσαμε στην εισαγωγή είναι ο εξής:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   vector< unsigned int > array;

   int x;
   cin >> x;
   while ( x >= 0 )
   {
      array.push_back( x );
      cin >> x;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   for ( int k(0); k < count; ++k )
      cout << array[k] << endl;
}
```

Όπως βλέπεις, αυτό είναι αρκετά απλούστερο από το αρχικό πρόγραμμα.

Το "`vector`" είναι *περιέχουσα κλάση* της STL και εδώ χρησιμοποιούμε ένα στιγμιότυπο της για περιεχόμενο "`unsigned int`".

Ο `array` είναι ένας δυναμικός πίνακας. Πρόσεξε ότι το μόνο σημείο που αναφερόμαστε στο μέγεθος του πίνακα είναι το "`array.size()`" με το οποίο ζητάμε από τον `array` να μας πει το μέγεθός του. Η "`array.push_back(x)`" αντιγράφει την τιμή της *x* στο τέλος του πίνακα. Αν το μέγεθος του πίνακα δεν είναι αρκετό φροντίζει να τον μεγαλώσει.

Το "`sort`" είναι ένα περίγραμμα συνάρτησης που περιμένει δύο προσεγγιστές που να ορίζουν μια περιοχή μέσα σε κάποιο περιέχον και ταξινομεί τις τιμές μέσα στην περιοχή. Η ταξινόμηση γίνεται με βάση τον τελεστή "`<`" που πρέπει να είναι ορισμένος για τον περιεχόμενο τύπο.

Τα "`array.begin()`" και "`array.end()`" είναι κλήσεις μεθόδων του `array` που μας δίνουν (σταθερούς) *προσεγγιστές* προς την αρχή και το τέλος του. Θυμίσου τις μεθόδους με τα ίδια ονόματα που είδαμε στην *SListT*.

26.1.1 Περιέχουσες Κλάσεις

Όπως λέγαμε και στο προηγούμενο κεφάλαιο (§25.7) «*Περιέχον (container) ή περιέχουσα κλάση (container class) είναι περίγραμμα για κλάση-συλλογή: κάθε αντικείμενο ενός στιγμιοτύπου της είναι συλλογή άλλων αντικειμένων.*» Στη συνέχεια θα χρησιμοποιούμε τον όρο «περιέχον» και για στιγμιότυπα κάποιας περιέχουσας κλάσης.

Μπορούμε να δούμε τα περιέχοντα της STL χωρισμένα σε δύο κατηγορίες (Σχ. 26-1):

- **Ακολουθίες** (sequences) στις οποίες η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Στις ακολουθίες ανήκουν τα
  - *vector* (διάνυσμα). Με τη δήλωση "`vector<K> vt;`" ο *vt* δηλώνεται ως δυναμικός μονοδιάστατος πίνακας (διάνυσμα) με στοιχεία τύπου *K*.
  - *list*. Με τη δήλωση "`list<K> lt;`" η *lt* δηλώνεται ως λίστα με διπλή σύνδεση με στοιχεία τύπου *K*.
  - *deque* (*d*ouble *e*nded *que*ue, ουρά δύο άκρων). Μπορεί να γίνεται εισαγωγή και εξαγωγή από τα δύο άκρα.

  και οι προσαρμογείς
  - *priority_queue* (ουρά προτεραιότητας). Γενικώς, τα στοιχεία εισάγονται μαζί με βαθμό προτεραιότητας και εξάγεται πρώτο το στοιχείο με τη μεγαλύτερη προτεραιότητα. Στην STL πρέπει να οριστεί η διάταξη "<" (για την περιεχόμενη κλάση) και πρώτο εξάγεται το «μεγαλύτερο».
  - *queue* (ουρά) FIFO. Επιτρέπει εισαγωγή μόνον στο τέλος της ουράς και εξαγωγή μόνον από την αρχή. Μπορείς να τη δεις σαν ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *φθίνουσα* προτεραιότητα.
  - *stack* (στοίβα) LIFO. Επιτρέπει εισαγωγή και εξαγωγή μόνον από ένα άκρο. Μπορείς να τη θεωρήσεις ουρά προτεραιότητας όπου τα εισερχόμενα στοιχεία έχουν *αύξουσα* προτεραιότητα.
- **Συνειρμικά Περιέχοντα** (associative containers) στα οποία η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση κάποιο κλειδί. Αυτά είναι:
  - *set* (σύνολο).
  - *multiset* (πολυσύνολο).
  - *map* (απεικόνιση), σύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).
  - *multimap* (πολυαπεικόνιση), πολυσύνολο ζευγών (κλειδί, υπόλοιπα στοιχεία).

  Όλα τα παραπάνω περιέχοντα έχουν:
- Ερήμην δημιουργό.
- Δημιουργό αντιγραφής.
- Καταστροφέα.
- *Αντιγραφικό* τελεστή εκχώρησης.
- Μέθοδο *size()* που επιστρέφει το πλήθος στοιχείων της συλλογής.
- Μέθοδο *max_size()* που επιστρέφει το μέγιστο πλήθος στοιχείων που μπορεί να αποθηκευτεί στη συλλογή.
- Μέθοδο *empty()* που επιστρέφει τιμή `true` αν η περιεχόμενη συλλογή δεν έχει στοιχεία.
- Μέθοδο *swap()* για ανταλλαγή τιμών δύο περιεχόντων (του ίδιου τύπου).
- Μέθοδο *get_allocator()* που επιστρέφει αντίγραφο του παραχωρητή μνήμης του περιέχοντος.

Έχουν ακόμη τέσσερις μεθόδους που τις βλέπουμε στη συνέχεια.

Με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι

- "==", "!=" (το $a \neq b$ υπολογίζεται ως !(a == b)),
- "<", ">=" ((a >= b) ≡ !(a < b)),
- ">", "<=" ((a > b) ≡ (b < a)).

Η σύγκριση "<" γίνεται *λεξικογραφικώς* (§22.7.3). Για να γίνει αυτό θα πρέπει να έχουμε (εκτός από τον "==") και τον "<" για την περιεχόμενη κλάση.

Κάθε μια από τις περιέχουσες κλάσεις έχουν τις δικές τους *κλάσεις προσεγγιστών*. Πρώτη η *iterator* που είναι σαν αυτήν που είδαμε στην *SListT*. Και αυτή έχει επιφορτωμένους τους "++", "*", "->", "==", "!=". Ακόμη, έχει επιφορτωμένον και τον "--": αυτοί οι προσεγγιστές είναι **αμφίδρομοι** (bidirectional) ενώ της *SListT* ήταν *μονόδρομοι*.

Η δεύτερη κλάση είναι η *const_iterator*. Αν δηλώσεις

```
C<T>::const_iterator it;
```

δεν έχεις δικαίωμα να αλλάξεις την τιμή του "*it".

Δύο μέθοδοι μας δίνουν την αρχή και το τέλος των περιεχομένων της περιέχουσας κλάσης:

- Μέθοδος *begin()*: που επιστρέφει προσεγγιστή προς το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
iterator begin();
const_iterator begin() const;
```

- Μέθοδος *end()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) μετά το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
iterator end();
const_iterator end() const;
```

(Η δεύτερη μορφή –*const_iterator*– για αντικείμενα-συλλογές "const".) Αν κάποιο περιέχον *a* είναι *κενό* –η *a*.*empty()* επιστρέφει true– τότε έχουμε *a*.*begin()* == *a*.*end()*.

Φυσικά, θα πεις: το «*πρώτο στοιχείο*» και η «*πρώτη θέση(!) μετά το τελευταίο στοιχείο*» μπορεί να έχουν νόημα για πίνακα, μπορεί να έχουν νόημα και για λίστα (όπως είδαμε στα προηγούμενα κεφάλαια, π.χ. §25.7.1)· τι νόημα μπορεί να έχουν για το σύνολο; Θα κατάλάβεις στη συνέχεια…

Για οποιαδήποτε περιέχουσα κλάση C της STL, με τη:

```
for ( C<T>::iterator it( co.begin() ); it != co.end(); ++it )
{ /* . . . */ }
```

μπορείς να περάσεις από όλα τα περιεχόμενα αντικείμενα τύπου *T* του αντικειμένου-συλλογή *co*. Αν δεν θέλεις να αλλάζεις τις τιμές των περιεχομένων μπορείς να χρησιμοποιήσεις προσεγγιστή "C<T>::const_iterator" ώστε να κάνεις το πρόγραμμά σου κάπως ταχύτερο.

Ας πούμε ότι η:

```
list< int > il;
```

έχει περιεχόμενο τους 0, 1, 2, 3, 4. Οι

```
for ( list<int>::iterator it( il.begin() );
      it != il.end(); ++it )
   cout << *it << "  ";
cout << endl;
```

θα δώσουν:

```
0  1  2  3  4
```

Δύο άλλες κλάσεις προσεγγιστών είναι οι *reverse_iterator* και *const_reverse_iterator*. Με τέτοιους προσεγγιστές διασχίζεις τη συλλογή από το τέλος προς την αρχή. Δύο μέθοδοι σου δίνουν αρχή και τέλος για τέτοια διάσχιση:

- Μέθοδος *rbegin()*: που επιστρέφει προσεγγιστή προς το τελευταίο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

- Μέθοδος *rend()*: που επιστρέφει προσεγγιστή προς την πρώτη θέση(!) πριν το πρώτο στοιχείο της περιεχόμενης συλλογής.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Οι εντολές

```
for ( list<int>::reverse_iterator rit( il.rbegin() );
      rit != il.rend(); ++rit )
   cout << *rit << "  ";
cout << endl;
```

για τη λίστα που είδαμε παραπάνω θα δώσουν:

```
4  3  2  1  0
```

Πρόσεξε ότι διασχίζουμε τη λίστα με την "`++rit`".

Αυτά προς το παρόν για τις περιέχουσες κλάσεις. Στη συνέχεια θα δούμε και άλλα που ισχύουν ειδικότερα για συγκεκριμένες κατηγορίες τους.

Υπάρχουν όμως και ορισμένες απαιτήσεις από τον τύπο της *περιεχόμενης* κλάσης. Στην §25.7.2 συνοψίσαμε μερικές από αυτές που επισημάναμε γράφοντας την περιέχουσα κλάση *SListT*:

- ***Ο τύπος περιεχομένου K θα πρέπει να έχει ερήμην δημιουργό.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει σωστό δημιουργό αντιγραφής.***
- ***Θα πρέπει να υπάρχει σωστός (αντιγραφικός) τελεστής εκχώρησης για τον περιεχόμενο τύπο K.***
- ***Ο τύπος περιεχομένου K θα πρέπει να έχει καταστροφέα.***

  Για ειδικές περιπτώσεις εμφανίζονται και άλλες απαιτήσεις. Ας πούμε,
- ***Αν χρειάζονται αναζητήσεις τιμών ο τύπος περιεχομένου K θα πρέπει να έχει τη δυνατοτητα για συγκρίσεις.***

26.1.2 Περί Προσεγγιστών...

Πριν προχωρήσουμε στην παρουσίαση των περιεχουσών κλάσεων της STL θα πούμε λίγα ακόμη για τους προσεγγιστές.

Η έννοια του προσεγγιστή είναι γενίκευση της έννοιας του βέλους. Με ποιον στόχο; Τη χρήση πάγιων τεχνικών και αλγορίθμων των μονοδιάστατων πινάκων σε άλλες δομές δεδομένων.

Αρχικώς, στην §22.9, γνωρίσαμε τον *πρόσθιο προσεγγιστή*. Σε αυτόν επιφορτώνουμε τους τελεστές "`*`", "`->`", "`++`", "`==`", "`!=`".

Ο *αμφίδρομος προσεγγιστής*, που είδαμε πιο πάνω, έχει όλα τα χαρακτηριστικά του *πρόσθιου προσεγγιστή* αλλά σε αυτόν επιφορτώνουμε και τον "`--`".

Στη συνέχεια θα δούμε τους **προσεγγιστές τυχαίας πρόσβασης** (random access iterators) που δεν έχουν να ζηλέψουν κάτι από τα βέλη: έχουν όλα τα χαρακτηριστικά του αμφίδρομου προσεγγιστή και επιπλέον αριθμητική σαν αυτήν των βελών (§12.3): "*προσεγγιστής θ ακέραιος*" όπου θ κάποιος από τους "`+`", "`-`", "`+=`", "`-=`" ή "*προσεγγιστής* – *προσεγγιστής*" (που μας δίνει την «απόσταση» των στοιχείων που δείχνουν οι δύο προσεγγιστές).

Ας δούμε τώρα μια τρίτη μορφή του παραδείγματός μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <iterator>

using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
   sort( array.begin(), array.end() );
   size_type count( array.size() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   for ( int k(0); k < count; ++k )
      *coutIt = array[k];
   cout << endl;
}
```

και ένα παράδειγμα εκτέλεσης:

```
19  0  17  3  11  5  7 e
0  3  5  7  11  17  19
```

Βάζοντας στο πρόγραμμα την "`#include <iterator>`" μπορούμε να χρησιμοποιήσουμε το *ostream_iterator* που είναι ένα περίγραμμα που μπορεί να μας δώσει στιγμιότυπα για οποιονδήποτε τύπο *T* για τον οποίον υπάρχει επιφορτωμένος ο "`<<`" προς μορφοποιημένο αρχείο. Ο *coutIt* είναι ένας προσεγγιστής *ostream*. Με τη δήλωση του *coutIt* λέμε ότι θα είναι ένας προσεγγιστής που θα γράφει τιμές τύπου `unsigned int` στο *cout* και μετά από κάθε τιμή θα «γράφει» δύο διαστήματα (`"  "`). Πώς γράφει μια τιμή *v*; Η εντολή: "`*coutIt = v;`" Ισοδυναμεί με "`cout << v << "  ";`". Έτσι, η **for** του προγράμματος βγάζει το αποτέλεσμα που είδες παραπάνω.

Ο *ostream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εξόδου** (output iterator). Ένας προσεγγιστής εξόδου έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και κατάστροφέα. Ακόμη μπορεί να «αυξηθεί» (π.χ. με τον "`++`").
- Με την αποπαραπομπή του παίρνουμε *τιμή-l* (§11.3). Στην πραγματικότητα η αποπαραπομπή ενός προσεγγιστή εξόδου εμφανίζεται πάντοτε στο αριστερό μέρος μιας εκχώρησης όπως φαίνεται και στο παράδειγμά μας ("`*coutIt = array[k];`").

Όπως καταλαβαίνεις και οι τρεις κατηγορίες προσεγγιστών που είδαμε –πρόσθιοι, αμφίδρομοι, τυχαίας πρόσβασης– έχουν τα χαρακτηριστικά του προσεγγιστή εξόδου.

Όπως πιθανότατα μαντεύεις, υπάρχουν και *προσεγγιστές istream*. Έτσι μπορούμε να διαβάσουμε από κάποιο *istream* ως εξής:

```
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;

   while ( cinIt != cinEoStream )
   {
      array.push_back( *cinIt );
      ++cinIt;
   } // while
```

Ο *cinIt* είναι ένας προσεγγιστής κλάσης `istream_iterator<unsigned int>`: Διαβάζει τιμές τύπου `unsigned int` από ένα ρεύμα *istream*, στην περίπτωσή μας από το *cin*. Δηλώνοντας έναν *istream_iterator* χωρίς ορίσματα –όπως τον *cinEoStream*– παίρνεις έναν προσεγγιστή που όμως «δείχνει μετά το τέλος του ρεύματος τιμών `unsigned int`» και χρησιμοποιείται για τον τερματισμό της ανάγνωσης.

Η τιμή που διαβάζουμε δίνεται με την αποπαραπομπή του προσεγγιστή (`*cinIt`). Ο προσεγγιστής διαβάζει με τον "`>>`". Προχωρούμε στην επόμενη τιμή με τον τελεστή "`++`".

Αν φτάσουμε σε τέλος αρχείου ή σε τέλος ρεύματος (δηλαδή –στην περίπτωσή μας– τιμή που δεν είναι `unsigned int`) η τιμή του *cinIt* θα γίνει ίση με αυτήν του *cinEoStream*.

Ο *istream_iterator* είναι το απλούστερο παράδειγμα **προσεγγιστή εισόδου** (input iterator) που έχει τα εξής χαρακτηριστικά:

- Έχει δημιουργό αντιγραφής, (αντιγραφικό) τελεστή εκχώρησης και κατάστροφέα. Αν δεν έχει φτάσει στο τέλος ρεύματος, μπορεί να «αυξηθεί» (π.χ. με τον "`++`"). Ακόμη, μπορεί να συγκριθεί με τον "`==`" και τον "`!=`".
- Με την αποπαραπομπή του παίρνουμε *τιμή-r* (μπορεί να εμφανιστεί μόνον στο δεξιό μερος της εκχώρησης και όχι στο αριστερό).

Τα χαρακτηριστικά του προσεγγιστή εισόδου τα βρίσκουμε και στους προσεγγιστές που είδαμε: εξόδου, πρόσθιους, αμφίδρομους και τυχαίας πρόσβασης.

Βάζοντας στο πρόγραμμά σου "`#include <iterator>`" μπορείς να χρησιμοποιήσεις το περίγραμμα συνάρτησης

```
template < typename InputIterator, typename Distance >
void advance( InputIterator& it, Distance n );
```

που δίνει τη λειτουργικότητα του "`+=`" σε οποιονδήποτε προσεγγιστή εισόδου. Ο *Distance* τι τύπος είναι; Κατ' αρχήν είναι ακέραιος τύπος. Αν ο *InputIterator* είναι τύπος προσεγγιστή αμφίδρομου ή τυχαίας πρόσβασης ο *Distance* μπορεί να είναι και ο `int`. Αλλιώς πρέπει να είναι `unsigned int`.

Η "`advance(it, n);`"

- αν ο *it* είναι τυχαίας πρόσβασης, θα εκτελεσθεί ως "`it += n;`"
- αλλιώς θα εκτελεσθεί με επαναληπτική (*n* φορές) δράση του "`++it;`" ή "`--it;`".

26.1.3 Αλγόριθμοι

Έχουμε ήδη μάθει ότι βάζοντας στο πρόγραμμά μας "`#include <algorithm>`" μπορούμε να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων:

- (*std*::)*max()*, (*std*::)*min()*,
- (*std*::)*swap()*,
- (*std*::)*find()*,
- (*std*::)*sort()*.

Στη συνέχεια θα δούμε μερικά ακόμη, αλλά δεν θα παραθέσουμε και τα πάντα. Ψάξε τα εγχειρίδια της C++ που χρησιμοποιείς για να δεις τι υπάρχει.

Εδώ θα δούμε άλλη μια μορφή του παραδείγματός μας χρησιμοποιώντας το περίγραμμα:

```
template< class InputIterator, class OutputIterator >
OutputIterator copy( InputIterator first, InputIterator last,
                     OutputIterator result );
```

που αντιγράφει τα στοιχεία

από την περιοχή `[` *first*, *last* `)` στην περιοχή `[` *result*, *result* + (*last* - *first*) `)`.

Κάνει δηλαδή τις εκχωρήσεις "`*result = *first`", "`*(result+1) = *(first +1)` ", κ.ο.κ. με αυτήν τη σειρά. Επιστρέφει ως τιμή το *result* + (*last* - *first*)· δηλαδή έναν προσεγγιστή προς μια «θέση» μετά το τελευταίο στοιχείο που ήλθε από την αντιγραφή.

Δες πώς γράφεται το πρόγραμμά μας:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
```

```
using namespace std;

int main()
{
   vector< unsigned int > array;
   istream_iterator< unsigned int > cinIt( cin );
   istream_iterator< unsigned int > cinEoStream;
   back_insert_iterator< vector<unsigned int> > dest( array );
   copy( cinIt, cinEoStream, dest );
   sort( array.begin(), array.end() );
   ostream_iterator< unsigned int > coutIt( cout, "  " );
   copy( array.begin(), array.end(), coutIt );
   cout << endl;
}
```

Πρόσεξε πώς γίνεται η έξοδος των στοιχείων: με αντιγραφή όλων των στοιχείων του πίνακα –αυτά περιλαμβάνονται στην περιοχή [*array*`.`*begin*`()`, *array*`.`*end*`()`)– στο ρεύμα *cout*. Αυτό είναι το εύκολο.

Η δυσκολία υπάρχει στην ανάγνωση. Θα μπορούσαμε να διαβάσουμε τα στοιχεία με την *copy()*; Να γράψουμε δηλαδή κάτι σαν:

```
   copy( cinIt, cinEoStream, array.begin() );
```

Όχι! Εδώ έχουμε πρόβλημα: Ο *array* δεν έχει την απαιτούμενη μνήμη και –γενικώς– δεν ξέρουμε πόσα είναι τα στοιχεία για να την κρατήσουμε πριν ζητήσουμε την αντιγραφή.

Η λύση είναι η εξής:

```
   back_insert_iterator< vector<unsigned int> > dest( array );
   copy( cinIt, cinEoStream, dest );
```

Ο *back_insert_iterator* είναι ένας *προσαρμογέας*: από τη μια είναι ένας προσεγγιστής, όπως τον θέλει η *copy()*, από την άλλη φροντίζει να παίρνει την απαραίτητη μνήμη ώστε να γίνεται η εισαγωγή μιας τιμής στον *array* χωρίς πρόβλημα.

Πριν προχωρήσουμε, αξίζει να «θαυμάσουμε» αυτή τη μορφή του προγράμματος που μοιάζει με ψευδοκώδικα! copy από την είσοδο – sort – copy στην έξοδο! Ή, αλλιώς: έχουμε προγραμματισμό σε ψηλότερο επίπεδο!

Παρατηρήσεις:▶

1. Η STL έχει περιγράμματα συναρτήσεων που δρουν σε *περιοχές* ενός περιέχοντος (ή σε *ολόκληρο* περιέχον). Δηλαδή, για ένα πρόβλημα σαν αυτό που είδαμε εδώ, η αντιπροσωπευτική λύση είναι η τελευταία και όχι οι προηγούμενες με τις **while** και **for**.
2. Όπως χρησιμοποιείς τα περιέχοντα της STL μπορείς να χρησιμοποιείς και τα περιέχοντα της C++ –δηλαδή τους πίνακες– και περιέχοντα δικά σου. Δες μια άλλη μορφή του παραδείγματος που είδαμε πιο πάνω:

```
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
   int  array[100];
   int* pEnd;

   pEnd = copy( istream_iterator<int>(cin), istream_iterator<int>(),
                array );
   sort( array, pEnd );
   copy( array, pEnd, ostream_iterator<int>(cout, "\n") );
}
```

Εδώ, περιέχον είναι ένας συνήθης πίνακας. Ως προσεγγιστές χρησιμοποιούμε δύο συμβατικά βέλη, τα `array` και `pEnd`. Κατά τα άλλα χρησιμοποιούμε τα περιγράμματα *copy()* και *sort()*.◀

Πριν προχωρήσουμε είναι καλό να ρίξουμε μια ματιά στις προδιαγραφές των περιγραμμάτων που ήδη είδαμε και χρησιμοποιούμε.

Ξεκινούμε από τα *min()* και *max()* που δίνονται με δύο μορφές:

```
template< class T >
const T& min( const T& a, const T& b );
template< class T, class Compare >
const T& min( const T& a, const T& b, Compare comp );
```

και

```
template< class T >
const T& max( const T& a, const T& b );
template< class T, class Compare >
const T& max( const T& a, const T& b, Compare comp );
```

Για τις απλές περιπτώσεις ο τύπος *T* θα πρέπει να έχει

- τον τελεστή "`<`" και
- δημιουργό αντιγραφής.

Έτσι, για παράδειγμα η *min()* μπορεί να είναι:

```
template < class T >
inline const T& min( const T& a, const T& b )
{  return b < a ? b : a;  }
```

Οι μορφές με τις τρεις παραμέτρους μπορεί να χρησιμοποιηθούν όταν ο *T* δεν έχει τον "`<`" ή θέλουμε να κάνουμε σύγκριση με άλλον τρόπο· αρκεί να δοθεί η *comp()* που μας λέει ποια από δύο τιμές *T* είναι η «μικρότερη» ή «προηγείται». Στην περίπτωση αυτήν η *min()* μπορεί να είναι:

```
template < class T, class Compare >
inline const T& min( const T& a, const T& b, Compare comp )
{  return comp(b, a) ? b : a;  }
```

Επομένως η *Compare* είναι ένα συναρτησοειδές, δηλαδή κλάση στην οποίαν έχουμε επιφορτώσει τον τελεστή "`()`" ώστε να δέχεται δύο παραμέτρους τύπου *T* και να επιστρέφει `true` αν η πρώτη είναι η «μικρότερη».

Αν έχεις να υπολογίσεις τον ελάχιστο από δύο ακέραιους χρησιμοποιείς την απλή μορφή: "`min( i1, i2 )`".

Αν όμως έχεις να συγκρίνεις τα `"abc"` και `"aggr"` θα χρειαστείς τη δεύτερη.[1] Για να τη χρησιμοποιήσεις έχεις δύο επιλογές:

- Να γράψεις κλάση που να ταιριάζει με την "`class Compare`":

```
struct CStrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {  return strcmp(cs1, cs2) < 0;  }
}; // CStrLT
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLT)`" όπου το *cstrLT* είναι (συναρτησιακό) αντικείμενο κλάσης *CStrLT*.

- Να γράψεις συνάρτηση (κατηγόρημα):

```
bool cstrLTf( const char cs1[], const char cs2[] )
{  return strcmp(cs1, cs2) < 0;  }
```

  και στη συνέχεια να δώσεις "`min("abc", "aggr", cstrLTf)`".

---

[1] Βέβαια μπορείς να βολευτείς με την απλή μορφή αν γράψεις "`min( string("abc"), string( "aggr") )`".

Παρόμοια ισχύουν και για τη *max()* αλλά προσοχή: μη θεωρήσεις ότι εδώ θα πρέπει να ασχοληθείς με τον "`>`"· και πάλι ο "`<`" θα κάνει τη δουλειά μας. Για παράδειγμα η "`max("abc", "aggr", cstrLTf)`" θα μας δώσει `"aggr"`.

Ας έλθουμε τώρα στο περίγραμμα:

```
template< class InputIterator, class T >
InputIterator find( InputIterator first, InputIterator last,
                    const T& value );
```

Στον τύπο *T* πρέπει να είναι δυνατές οι συγκρίσεις "`==`" και "`!=`".

Πάντως υπάρχει και το περίγραμμα *find_if()* που είναι γενίκευση του *find()*:

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Μας επιστρέφει προσεγγιστή *it* προς την πρώτη τιμή για την οποίαν το `pred(*it)` θα επιστρέψει `true`. Αν δεν βρει τέτοια τιμή επιστρέφει *last*.

Για το *pred* έχουμε τις ίδιες επιλογές που είχαμε και για το *comp* προηγουμένως αλλά το *pred* είναι κατηγόρημα με μια παράμετρο.

Και η *sort()* έχει δύο μορφές:

```
template< class RandomAccessIterator >
void sort( RandomAccessIterator first, RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void sort( RandomAccessIterator first, RandomAccessIterator last,
           Compare comp );
```

Όπως λέει και το όνομα της κλάσης-παραμέτρου πρέπει να έχουμε προσεγγιστές τυχαίας πρόσβασης. Στην πρώτη περίπτωση η ταξινόμηση γίνεται με βάση τον τελεστή "`<`" ενώ στη δεύτερη με βάση το κατηγόρημα (δύο παραμετρων) *comp()* για το οποίο ισχύουν αυτά που είπαμε πιο πάνω.

Τώρα θα πεις, αφού μας δίνει εργαλείο για ταξινόμηση δεν θα μας δώσει και ένα εργαλείο για δυαδική αναζήτηση; Μας δίνει και όχι μόνον ένα.

```
template< class ForwardIterator, class T >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value );
template< class ForwardIterator, class T, class Compare >
bool binary_search( ForwardIterator first, ForwardIterator last,
                    const T& value, Compare comp );
```

Εδώ έχεις μια απογοήτευση: αν βρεθεί στο `[`*first, last*`)` κάποιος *it* τέτοιος που να έχουμε `*`*it* `==` *value* η *binary_search()* θα σου επιστρέψει `true`. Αυτό συνήθως δεν είναι αρκετό.

Ας δούμε μιαν άλλη συνάρτηση (δηλαδή *δύο* άλλες):

```
template< class ForwardIter, class T >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value );
template< class ForwardIter, class T, class Compare >
ForwardIter lower_bound( ForwardIter first, ForwardIter last,
                         const T& value, Compare comp );
```

Ο προσεγγιστής που επιστρέφεται –ας τον πούμε *rv*– δείχνει την πρώτη θέση στην οποία μπορεί να εισαχθεί η *value* χωρίς να χαλάσει η ταξινόμηση· δείχνει το πρώτο στοιχείο που δεν είναι μικρότερο από *value*. Δηλαδή για κάθε προσεγγιστή *it* στο `[`*first, rv*`)` έχουμε `*`*it* `<` *value*. Αν *rv* `!=` *last* για κάθε *it* στο `[`*rv, last*`)` έχουμε `*`*it* `>=` *value* (ακριβέστερα: `!(*`*it* `<` *value*`)`.)

Αν χρησιμοποιήσεις τη δεύτερη μορφή τότε στο `[`*first, rv*`)` έχουμε `comp(*it, value)` (`== true`) ενώ στο `[`*rv, last*`)` έχουμε `!comp(*it, value)`.

Αν η αναζητούμενη τιμή υπάρχει στο `[`*first, last*`)` *τότε θα τη δείχνει ο rv δηλαδή θα έχουμε:* `*`*rv* `==` *value* (ακριβέστερα: `!(*`*rv* `<` *value*`)` `&&` `!(`*value* `<` `*`*rv*`)`.)

Παρατηρήσεις:▶

1. Και οι τέσσερις συναρτήσεις έχουν πολυπλοκότητα ανάλογη του *log*(*last–first*).

2. Η *comp* πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.
3. Υπάρχει και ένα ζεύγος περιγραμμάτων συναρτήσεων *upper_bound()* που επιστρέφουν προσεγγιστή προς τη μικρότερη τιμή που είναι μεγαλύτερη από αυτήν που ψάχνουμε.◀

26.1.3.1 Τρεις Συναρτήσεις για «τα Πάντα»

Πριν προχωρήσουμε στα συναρτησιακά αντικείμενα θα πούμε λίγα λόγια για τρία περιγράμματα συναρτήσεων που *δρουν σε όλα τα αντικείμενα μιας περιοχής*: πρόκειται για τις *for_each(), transform()* και *random_shuffle()*.

Η συνάρτηση

```
template< class InputIterator, class Function >
Function for_each( InputIterator first, InputIterator last, Function f );
```

σαρώνει την περιοχή [*first, last*) με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή του υπολογίζει το "`f(*it)`". Αν η *f()* δεν είναι `void` το αποτέλεσμα που επιστρέφει αγνοείται.

Τα αντικείμενα κλάσης *Function* έχουν επιφορτωμένο τον "`()`" (συναρτησιακά αντικείμενα) έτσι ώστε να δέχονται μοναδική παράμετρο τα αντικείμενα που προκύπτουν από την αποπαραπομπή προσεγγιστών κλάσης *InputIterator*. Το όνομα της κλάσης-παραμέτρου μας δείχνει ότι τα χαρακτηριστικά των προσεγγιστών: είναι *προσεγγιστές εισόδου*. Αν πάρουμε υπόψη μας ότι η αποπαραπομπή τους είναι *τιμή-r* καταλαβαίνουμε ότι *η "`f(*it)`" δεν μπορεί να αλλάζει την τιμή του αντικειμένου που δείχνει ο it*.

Η *for_each()* επιστρέφει το συναρτησιακό αντικείμενο *f* όπως αυτό διαμορφώθηκε μετά την εφαρμογή σε όλα τα αντικείμενα της περιοχής.

Το παρακάτω πρόγραμμα

```
#include <iostream>
#include <algorithm>

using namespace std;

template< typename T >
struct Print
{
   void operator()( T x )
   {  cout << x << ' ';  }
}; // Print

int main()
{
   int        intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   Print<int> pr;

   for_each( intArr, intArr + 7, pr );
   cout << endl;
}
```

θα δώσει:

```
34 72 -23 8 -11 31 41
```

Θα μπορούσαμε να πάρουμε τα ίδια αποτελέσματα με τις:

```
   for_each( intArr, intArr+7, printF<int> );
   cout << endl;
```

όπου

```
template< typename T >
void printF( T x )  {  cout << x << ' ';  }
```

Για να δεις πώς μπορούμε να χρησιμοποιήσουμε αυτό που επιστρέφει η *for_each()* αλλάζουμε λίγο την *Print* (SGI 1999):

```
template< typename T >
struct Print
```

```
{
   Print() :  count(0) {};
   void operator()( T x )
   {  cout << x << ' ';
      ++count;  }
   unsigned int count;
}; // Print
```

και τον τρόπο που την καλούμε:

```
   pr = for_each( intArr, intArr+7, pr );
   cout << endl << pr.count << " numbers printed" << endl;
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
7 numbers printed
```

Αν θέλεις να αλλάξεις «τα πάντα» θα χρησιμοποιήσεις ένα από τα δύο περιγράμματα:

```
template< class InputIter, class OutputIter,
          class UnaryOperation >
OutputIterator transform( InputIter first, InputIter last,
                          OutputIter result, UnaryOperation op );
template< class InputIter1, class InputIter2,
          class OutputIter, class BinaryOperation >
OutputIterator transform( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, OutputIter result,
                          BinaryOperation binary_op );
```

Στα στιγμιότυπα της πρώτης μορφής σαρώνεται η περιοχή [*first, last*) με έναν προσεγγιστή (*InputIterator*) *it* και για κάθε τιμή υπολογίζεται το **op(*it)** που φυλάγεται σε διαδοχικές θέσεις ξεκινώντας από αυτήν που δείχνει ο *result*. Ο *result* μπορεί να δείχνει την ίδια θέση που δείχνει και ο *first*. Επιστρέφει προσεγγιστή προς την πρώτη θέση μετά την τελευταία όπου αποθηκεύτηκε αποτέλεσμα.

Ένα απλό παράδειγμα χρήσης της πρώτης μορφής είναι το παρακάτω:

```
#include <iostream>
#include <algorithm>

using namespace std;

template< class T >
struct Sqr
{
   T operator()( T x )  {  return x*x;  }
}; // Sqr

int main()
{
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   int      intArr1[7];
   Sqr<int> sqr;

   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   int* res1( transform(intArr, intArr+7, intArr1, sqr) );
   copy( intArr1, intArr1+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res1-intArr1) << endl;

   int* res( transform(intArr, intArr+7, intArr, sqr) );
   copy( intArr, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl << (res-intArr) << endl;
}
```

Αποτέλεσμα:

```
34 72 -23 8 -11 31 41
1156 5184 529 64 121 961 1681
7
1156 5184 529 64 121 961 1681
```

7

Το συναρτησιακό αντικείμενο *Sqr*[2] υπολογίζει και επιστρέφει το τετράγωνο της μοναδικής παραμέτρου του.

Η πρώτη κλήση της *transform()* έχει ως αποτέλεσμα να υπολογιστούν τα τετράγωνα των στοιχείων του `intArr` και να φυλαχτούν στα αντίστοιχα στοιχεία του `intArr1`. Ο προσεγγιστής (στην περίπτωσή μας απλό βέλος) που επιστρέφει (*res1*) απέχει από την αρχή του `intArr1` 7 θέσεις τύπου `int`.

Στη δεύτερη κλήση της *transform()* κάθε τετράγωνο που υπολογίζεται αντικαθιστά το αρχικό στοιχείο του `intArr`. Ο προσεγγιστής που επιστρέφει (*res*) απέχει από την αρχή του `intArr` 7 θέσεις τύπου `int`.

Πρόσεξε τώρα τη δεύτερη μορφή: το συναρτησιακό αντικείμενο που έχουμε εδώ έχει δύο παραμέτρους και δρα στα αντίστοιχα στοιχεία δύο περιοχών με ίσα μήκη. Το αποτέλεσμα κατά τη φύλαξή του μπορεί να αντικαταστήσει είτε την πρώτη περιοχή είτε τη δεύτερη (είτε καμία από τις δύο). Στις παραμέτρους υπάρχει μια «ασυμμετρία»:

- Για την πρώτη περιοχή μας δίνεται αρχή και τέλος: `[`*first1*, *last1*`)`.
- Για τη δεύτερη περιοχή μας δίνεται μόνο η αρχή *first2* και το τέλος συνάγεται από το ότι θα πρέπει να έχει το ίδιο μήκος με την πρώτη.

Αν στο προηγούμενο παράδειγμα ορίσουμε επί πλέον

```
template< typename T >
struct Sum
{
   T operator()( T x, T y )  {  return x + y;  }
};
```

και δηλώσουμε:

```
   Sum<int> sum;
   int      intArr2[7];
```

τότε οι:

```
   transform( intArr, intArr+7, intArr1, intArr2, sum );
   copy( intArr2, intArr2+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
```

θα μας δώσουν:

```
1190 5256 506 72 110 992 1722
```

Όπως βλέπεις, το 1190 είναι 34+1156, 5256 `==` 72+5184 κ.ο.κ. Δηλαδή, η *sum* καλείται 7 φορές για να υπολογίσει τα `intArr[k] + intArr1[k]`. Το αποτέλεσμα φυλάγεται στο `intArr2[k]`. Θα μπορύσαμε, αντί για `intArr2`, να είχαμε βάλει `intArr` ή `intArr1`.

Η τρίτη «συνάρτηση για τα πάντα» είναι η *random_shuffle()* που έχει δύο μορφές:

```
template< class RandomAccessIter >
void random_shuffle( RandomAccessIter first, RandomAccessIter last);
template< class RandomAccessIter, class RandomNumberGenerator >
void random_shuffle( RandomAccessIter first, RandomAccessIter last,
                     RandomNumberGenerator& rand );
```

Αυτή ανακατεύει με τυχαίο πρόπο τις τιμές που δείχνουν οι προσεγγιστές της περιοχής `[`*first1*, *last1*`)`. Στη δεύτερη μορφή έχεις τη δυνατότητα να ορίσεις δική σου γεννήτρια τυχαίων αριθμών με μορφή συναρτησιακού αντικειμένου *rand*, με επιφορτωμένο τον "`()`", τέτοια ώστε η "`rand(n)`" να επιστρέφει τυχαίο ακέραιο στο `[`0`..`*n*`)`.

Ως παράδειγμα δες το ανακάτεμα της τράπουλας:

```
#include <iostream>
#include <algorithm>

using namespace std;
```

[2] Δεν είναι απαραίτητο να είναι περίγραμμα.

```
int main()
{
   char* cardDeck[52] = { "S2","S3","S4","S5","S6","S7","S8","S9","S10","SJ",
                                                            "SQ","SK","SA",
                          "H2","H3","H4","H5","H6","H7","H8","H9","H10","HJ",
                                                            "HQ","HK","HA",
                          "D2","D3","D4","D5","D6","D7","D8","D9","D10","DJ",
                                                            "DQ","DK","DA",
                          "C2","C3","C4","C5","C6","C7","C8","C9","C10","CJ",
                                                            "CQ","CK","CA" };
   random_shuffle( cardDeck, cardDeck+52 );
   for ( int k(0); k < 52; ++k )
   {
      cout << cardDeck[k] << ' ';
      if ( k%13 == 12 )  cout << endl;
   }
}
```

Αποτέλεσμα:[3]

```
DK SK D5 HQ H7 D3 C4 CQ H9 H4 S6 D8 C10
S7 C9 C2 D9 DA H3 D10 CJ S10 H10 CA S9 H8
C3 SA C6 SJ H2 H5 HJ D6 D7 CK S2 C7 DJ
HA D2 H6 C8 S4 C5 S8 S3 SQ DQ HK S5 D4
```

26.1.4 Συναρτησιακά Αντικείμενα

Στη συνέχεια θα δούμε το περίγραμμα κλάσης "*set*" που είναι:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

Αφού στα περιγράμματα κλάσεων δεν μπορούμε να έχουμε επιφόρτωση, όπως έχουμε στα περιγράμματα συναρτήσεων, αλλά μπορούμε να έχουμε ερήμην «τιμές» των παραμέτρων του περιγράμματος, αντί να έχουμε δύο περιγράμματα "*set*", όπως έχουμε δύο περιγράμματα *sort()*, έχουμε αυτό το "`class Compare = less<Key>`" που μας λέει ότι αν γράψω "`set<T>`" εννοώ ότι οι συγκρίσεις θα γίνονται με το συναρτησιακό αντικείμενο `less<T>` όπου (§25.6.3):

```
template < typename T >
struct less
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

Βάζοντας στο πρόγραμμά σου "`#include <functional>`" έχεις τη δυνατοτητα να χρησιμοποιείς συναρτησιακά αντικείμενα σαν το παραπάνω και τα άλλα παρόμοια περιγράμματα που υπάρχουν εκεί.

Όλα ξεκινούν με τα[4]

```
template < class Arg, class Result >
struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type;
}; // unary_function

template < typename Arg1, typename Arg2, typename Result >
```

[3] Πού είναι η τράπουλα; Βλέπε το "Sx" ως "♠x", το "Hx" ως "♥x", το "Dx" ως "♦x" και το "Cx" ως "♣x".

[4] Στο C++11 αποθαρρύνεται η χρήση των *unary_function*, *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

```
struct binary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
}; // binary_function
```

Τα άλλα περιγράμματα κληρονομούν αυτά τα δύο, για παράδειγμα:

```
template < class T >
struct negate : public unary_function< T, T >
{
   T operator() ( const T& x ) const { return -x; }
};

template < class T >
struct plus : public binary_function<T, T, T>
{
   T operator() ( const T& x, const T& y ) const { return x + y; }
};

template < typename T >
struct less : public binary_function< T, T, bool >
{
   bool operator()( const T& x, const T& y ) const
   {  return x < y;  }
};
```

- Το *binary_function* κληρονομούν τα
  - *plus()* (“`+`”), *minus()* (“`-`”), *multiplies()* (“`*`”), *divides()* (“`/`”), *modulus()* (“`%`”)
  - *equal_to()* (“`==`”), *not_equal_to()* (“`!=`”), *greater()* (“`>`”), *less()* (“`<`”), *greater_equal()* (“`>=`”), *less_equal()* (“`<=`”)
  - *logical_and()* (“`&&`”), *logical_or()* (“`||`”)
- Το *unary_function* κληρονομούν τα
  - *negate()* (“`-`”)
  - *logical_not()* (“`!`”)

Γυρνώντας στο παράδειγμα για τη δεύτερη μορφή της *transform()* βλέπουμε ότι δεν χρειάζεται να γράψουμε το περίγραμμα *Sum*. Βάζουμε στο πρόγραμμα την “`#include <functional>`”, αλλάζουμε τη δήλωση του *sum* σε:

```
   plus<int> sum;
```

και παίρνουμε τα ίδια αποτελέσματα.

Τα εργαλεία αυτά μας βοηθούν να χρησιμοποιούμε τις συναρτήσεις του `algorithm` και τα περιέχοντα της STL. Για τον λόγο αυτόν μας δίνονται και διάφοροι προσαρμογείς. Με ένα παράδειγμα ας δούμε εργαλεία που μας δίνονται ώστε να χρησιμοποιούμε συναρτησεις με δύο παραμέτρους εκεί που απαιτείται συνάρτηση με μια.

Παράδειγμα

Στο Project04, θέλουμε να εφοδιάσουμε την κλάση *CourseCollection* με μια μέθοδο:

```
Course* CourseCollection::findDep( string code )
```

που θα μας επιστρέφει βέλος προς στοιχείο (τύπου *Course*) που έχει ως προαπαιτούμενο (*prereq*) το στοιχείο με κωδικό *code*. Αν δεν υπάρχει τέτοιο στοιχείο θα επιστρέφει “`0`” (*NULL*).

Ένας τρόπος να λύσουμε το πρόβλημα είναι αυτός που έχουμε στη μέθοδο *delete1-Course()*· εδώ όμως θέλουμε λύσουμε το πρόβλημα χρησιμοποιώντας το

```
template< class InputIterator, class Predicate >
InputIterator find_if( InputIterator first, InputIterator last,
                       Predicate pred );
```

Το στιγμιότυπο που μας ενδιαφέρει θα είναι:

```
Course* find_if( ccArr, ccArr+ccNOfCourses, pred???? );
```

Τι θα είναι εκείνο το "`pred????`"; Εμείς ψάχνουμε ένα στοιχείο *s* για το οποίο "`s.prereq == code`". Αυτό θα μπορούσαμε να το κωδικοποιήσουμε με ένα κατηγόρημα:

```
struct EqPrereq
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Αλλά εδώ έχουμε ένα κατηγόρημα με δύο παραμέτρους ενώ το "`pred????`" πρέπει να έχει μια παράμετρο που θα αντικαθίσταται με τα `ccArr[0]`, `ccArr[1]` κ.ο.κ. Αυτό μπορεί να προκύψει από το *EqPrereq* ως εξής:

`bind2nd(EqPrereq(),code)`

που σημαίνει: συνάρτηση με μια παράμετρο που προκύπτει από την *EqPrereq()* αν βάλουμε τη δεύτερη παράμετρο ίση με "`code`".

Για να δουλέψει η *bind2nd()* θα πρέπει το (πρώτο) όρισμά της να κληρονομεί ένα στιγμιότυπο της *binary_function*. Γράφουμε λοιπόν:

```
struct EqPrereq :  binary_function< Course, string, bool >
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Η συνάρτηση που θέλουμε να γράψουμε θα είναι:

```
Course* CourseCollection::findDep( string code )
{
   Course* k;
   k = find_if( ccArr, ccArr+ccNOfCourses, bind2nd(EqPrereq(),code) );
   if ( k == ccArr+ccNOfCourses )  k = 0;
   return k;
} // findDep
```

❖❖❖

Εκτός από το περίγραμμα *bind2nd()* υπάρχει και το *bind1st()* που μας επιτρέπει να «καρφώσουμε» το πρώτο όρισμα. Τα δύο περιγράμματα στηρίζονται στα περιγράμματα-προσαρμογείς *binder2nd* και *binder1st* αντιστοίχως. Να επαναλάβουμε ότι για να χρησιμοποιήσουμε τα περιγράμματα συναρτήσεων σε μια συνάρτηση *f()* με δύο παραμέτρους θα πρέπει η αντίστοιχη συναρτησιακή κλάση *f* να κληρονομεί την *binary_function*.[5]

26.2 Ακολουθίες

Όπως είπαμε παραπάνω στις *ακολουθίες* η πρόσβαση σε κάποιο στοιχείο γίνεται με βάση τη θέση του. Έτσι:

- Ένα στιγμιότυπο της *vector* είναι κλάση μονοδιάστατων δυναμικών πινάκων. Αν έχουμε "`vector<K> vt`" τότε το "`vt[j]`" μας δίνει πρόσβαση στο αντίστοιχο στοιχείο του πίνακα με την ίδια ευκολία για οποιαδήποτε (νόμιμη) τιμή του "`j`".
- Ένα στιγμιότυπο της *list* είναι κλάση που κάθε της αντικείμενο είναι λίστα με διπλή σύνδεση. Αν έχουμε "`list<K> lt`" μπορείς να αρχίσεις να διασχίζεις την *lt* από την αρχή της (με τον *lt.begin()*) ή από το τέλος της (με τον *lt.end()*). Αν έχεις κάποιον προσεγγιστή *it* προς κάποιο στοιχείο της λίστας μπορείς να τον μετακινήσεις προς το επόμενο στοιχείο (με την "`++it`") ή προς το προηγούμενο (με την "`--it`") αν, φυσικά, υπάρχουν.

[5] Στο C++11 αποθαρρύνεται η χρήση των *bind2nd()*, *bind1st()*, *binder2nd*, *binder1st* όπως και της *binary_function* (που μπορεί να μην υπάρχουν σε επόμενες τυποποιήσεις).

- Ένα στιγμιότυπο της *deque* είναι κλάση που κάθε της αντικείμενο είναι ουρά δύο άκρων. Η λειτουργικότητα ενός τέτοιου αντικειμένου είναι παρόμοια με αυτήν ενός αντικειμένου *vector*, έχουμε όμως και τη δυνατότητα να εισάγουμε και να εξάγουμε αντικείμενα της περιεχόμενης κλάσης από την αρχή.

Όλες οι ακολουθίες `S<K>` έχουν –πέρα από αυτά που είπαμε για τα περιέχοντα γενικώς– και άλλες συναρτήσεις-μέλη που είναι:

- **Δημιουργοί πλήρωσης** (fill constructors).[6]

  ```
  explicit S<K>( size_type sz, const K& val = K() );
  ```

  Με τη δήλωση:

  ```
  S<K> a( n, c );
  ```

  –όπου $n \geq 0$ και *c* αντικείμενο κλάσης *K*– το *a* περιέχει *n* αντίγραφα του *c*. Για παράδειγμα, μετά την

  ```
  list< Date >  ld( 3, Date(2010, 1, 1) );
  ```

  η *ld* είναι λίστα με τρία αντίγραφα της "`1.1.2010`". Παρόμοια λίστα (χωρίς όνομα) δημιουργείται από την παράσταση:

  ```
  list< Date >( 3, Date(2010, 1, 1) )
  ```

  Με τη δήλωση:

  ```
  S<K> a( n );
  ```

  το *a* περιέχει *n* αντίγραφα του *K()*. Έτσι, αν δηλώσουμε:

  ```
  vector< Date >  vd( 5 );
  ```

  ο *vt* είναι πίνακας με πέντε αντίγραφα της "`1.1.1`". Παρόμοιος πίνακας (χωρίς όνομα) δημιουργείται και από την παράσταση: "`vector<Date>(5)`"

- **Δημιουργοί περιοχής** (range constructors).

  ```
  template < class InputIterator >
  S<K>( InputIterator first, InputIterator last );
  ```

  Με τη δήλωση:

  ```
  S<K> a( j, k );
  ```

  –όπου τα *j*, *k* είναι προσεγγιστές σε κάποιο άλλο περιέχον (ή απλά βέλη προς στοιχεία απλού πίνακα) και η `[j, k)` είναι μια μη κενή περιοχή με αντικείμενα κλάσης *K*– το *a* περιέχει αντίγραφα όλων των αντικειμένων της περιοχής. Για παράδειγμα, μετά τις

  ```
  int v0[10] = { 0,1,2,3,4,5,6,7,8,9 };
  int* p1( &v0[5] );  int* p2( &v0[8] );
  deque< int > dqt( p1, p2 );
  ```

  η *dqt* περιέχει 3 στοιχεία: τα "`5`", "`6`", "`7`".

  Ακόμη, στις τρεις «νέες» μορφές του παραδείγματος με το οποίο ξεκινήσαμε, μπορείς να προσθέσεις άλλη μια όπου η ανάγνωση των στοιχείων γίνεται με τη δήλωση του πίνακα: αφού δηλώσεις τους δύο *istream_iterator*s χρησιμοποίησε αυτόν τον δημιουργό για να δηλώσεις τον `array`:

  ```
    istream_iterator< unsigned int > cinIt( cin );
    istream_iterator< unsigned int > cinEoStream;
    vector< unsigned int > array( cinIt, cinEoStream );
    sort( array.begin(), array.end() );
// . . .
  ```

- Μέθοδοι για την **κεφαλή** (head) του περιέχοντος:

  ```
  K& front();
  const K& front() const;
  ```

  (η δεύτερη για αντικείμενα `const`.) Επιστρέφει (αναφορά προς) το πρώτο αντικείμενο (κεφαλή) του περιέχοντος. Με άλλα λόγια το: `a.front()` είναι ίσο με `*(a.begin())`.

---

[6] Παραλείπουμε την τρίτη παράμετρο (παραχωρητής μνήμης).

- Μέθοδοι εισαγωγής στοιχείων:

```
S<K>::iterator insert( S<K>::iterator pos, const K& val );
void insert( S<K>::iterator pos, size_type sz, const K& val );
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

  Η παράμετρος "`S<K>::iterator pos`" πρέπει να είναι προσεγγιστής στο περιέχον που κάνουμε την εισαγωγή.

  Η πρώτη μέθοδος εισάγει αντίγραφο της *val* πριν από το στοιχείο που δείχνει ο *pos* και αυξάνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που εισάχθηκε.

  Η δεύτερη εισάγει *sz* αντίγραφα της *val* πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *sz*.

  Η τρίτη (περίγραμμα) εισάγει ολόκληρη την περιοχή [*first*, *last*) πριν από το στοιχείο που δείχνει ο *pos*· αυξάνει το *size()* κατά *last - first*. Η προς αντιγραφή περιοχή μπορεί να βρίσκεται στο ίδιο ή σε άλλο αντικείμενο.

  Προσοχή: Η εισαγωγή στοιχείων σε μια ακολουθία μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας. Αυτό ισχύει και για το *vector* αφού με την εισαγωγή στοιχείου μπορεί να γίνει νέα παραχώρηση μνήμης. Έτσι, ο προσεγγιστής που επιστρέφει η πρώτη μορφή της *insert()* δεν είναι απαραίτητο να ισούται με τον (εισερχόμενο) *pos*.

- Μέθοδοι διαγραφής στοιχείων:

```
S<K>::iterator erase( S<K>::iterator pos );
S<K>::iterator erase( S<K>::iterator first,
                      S<K>::iterator last );
void clear();
```

  Η πρώτη μέθοδος διαγράφει το στοιχείο που δείχνει ο *pos* και μειώνει το *size()* κατά 1. Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που ακολουθούσε αυτό που διαγράφηκε.

  Η δεύτερη μέθοδος διαγράφει όλα τα στοιχεία της περιοχής [*first*, *last*). Επιστρέφει προσεγγιστή που δείχνει το στοιχείο που αρχικώς έδειχνε ο *last*.

  Η τρίτη διαγράφει όλα τα περιεχόμενα. Η `a.clear()` είναι ισοδύναμη με `a.erase( a.begin(), a.end() )`.

  Προσοχή: Και η διαγραφή στοιχείων ακολουθίας μπορεί να προκαλέσει την ακύρωση προσεγγιστών που έδειχναν στοιχεία της ακολουθίας.

- Μέθοδος αλλαγής μήκους

```
void resize( size_type sz, K val = K() );
```

  Δίνοντας "`vt.resize( n, kv );`"

  αν *n* < *vt*.*size*() είναι σαν να δίνεις

  "`vt.erase( vt.begin()+n, vt.end() );`"

  ενώ αν *n* > *vt*.*size*() είναι σαν δίνεις

  "`vt.insert( vt.end(), n-vt.size(), kv );`"

Τα τρία περιγράμματα ακολουθιών έχουν μερικές ακόμη ιδιότητες (που δεν υπάρχουν για τα επιπλέον περιγράμματα που προδιαγράφονται στο C++11):

- Είναι **αναστρέψιμα** (reversible) περιέχοντα. Όπως είπαμε παραπάνω, αυτό σημαίνει ότι έχουν τύπο *reverse_iterator* και τις μεθόδους *rbegin()*, *rend()*.
- Δίνουν εργαλεία για **εισαγωγή** (και **διαγραφή**) **στο τέλος** (back insertion).
  - Μέθοδο για να πάρουμε το τελευταίο στοιχείο του περιέχοντος (αν δεν είναι κενό):

    ```
    K& S<K>::back();
    ```

```
const K& S<K>::back();
```

Το "`a.back()`" είναι ισοδύναμο με το "`*(--a.end())`".

- Μέθοδο για εισαγωγή στο τέλος της συλλογής του περιέχοντος:

```
void S<K>::push_back( const K& );
```

Μετά την "`a.push_back( c );`" η "`a.back()`" μας δίνει το *c*.

- Μέθοδο για τη διαγραφή του τελευταίου στοιχείου της συλλογής του περιέχοντος (αν δεν είναι κενό):

```
void S<K>::pop_back();
```

26.2.1 Το Περίγραμμα "*vector*"

Πολλοί προγραμματιστές «βολεύονται» απλώς και μόνον με δυναμικούς πίνακες. Έτσι, «περιφρονούν» τις άλλες περιέχουσες κλάσεις της STL, χρησιμοποιούν μόνον τη *vector* (διάνυσμα) που γίνεται η ευρύτερα χρησιμοποιούμενη περιέχουσα κλάση της STL. «Βολεύονται» μόνο από τεμπελιά; Όχι!

♦ ***Το περίγραμμα "vector" είναι η μόνη περιέχουσα κλάση της STL[7] που εγγυάται αποθήκευση των στοιχείων σε συναπτές θέσεις της μνήμης.***

Έτσι, μπορεί κανείς να χρησιμοποιεί απλά βέλη –αλλά και συναρτήσεις– που έχουν γραφεί για συνήθεις (μονοδιάστατους) πίνακες για να χειριστεί πίνακες *vector*. Ας δούμε ένα

Παράδειγμα

Στην §12.3.5 είχαμε δει τη συνάρτηση

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

που υπολογίζει το άθροισμα των στοιχείων από `x[from]` μέχρι και `x[upto]` ενός πίνακα με *n* στοιχεία τύπου `double`. Μπορούμε να τη χρησιμοποιήσουμε για τον

```
   vector< double > vx;
```

Βεβαίως, και να πώς:

```
ut = vectorSum( &vx[0], vx.size(), 1, 2 );
```

Η αρχή του πίνακα βρίσκεται στη `&vx[0]`[8] και το πλήθος των στοιχείων του είναι `vx.size()`.

Το περίγραμμα *vector* δηλώνεται ως εξής:

```
template < class K, class Alloc = allocator<K> >
class vector;
```

Να δούμε τώρα τις δυνατότητες –για την ακρίβεια: τις μεθόδους– που προσφέρει η *vector* πέρα από αυτές που έχουν οι ακολουθίες γενικώς:

- Η πρόσβαση σε κάθε στοιχείο γίνεται (θυμίσου και τη *string*) με τον τελεστή "`[]`" και τη μέθοδο *at()*.

```
K& operator[]( size_type n );
const K& operator[]( size_type n ) const;
```

[7] Αυτό για τη C++03. Η STL της C++11 έχει και άλλες.

[8] Όπως θα δεις στη συνέχεια, στη *vector* επιφορτώνουμε τον "`[]`".

```
K& at( size_type n );
const K& at( size_type n ) const;
```

(Οι περιπτώσεις "`const`" για σταθερά αντικείμενα.) Όπως και στη *string*, η διαφορά έγκειται στο ότι η *at()* ελέγχει την τιμή του *n* και αν είναι ≥ *size()* ρίχνει εξαίρεση (*std*::)*out_of_range*.

- Όπως για τη *string* (§21.10.1) έτσι και για κάθε στιγμιότυπο του *vector* έχει νόημα η **χωρητικότητα** (capacity): το πλήθος αντικειμένων τύπου *K* που μπορούμε να αποθηκεύσουμε στη μνήμη που έχει ήδη δεσμευθεί για ένα αντικείμενο `vector<K>`. Μας τη δίνει η μέθοδος:

  ```
  size_type capacity() const;
  ```

  Προφανώς, για οποιοδήποτε αντικείμενο *a* τέτοιου τύπου θα έχουμε:

  `a.size() ≤ a.capacity()`

  Αν το `a.size()` πρέπει υπερβεί την τρέχουσα `a.capacity()` τότε θα πρέπει να γίνει νέα παραχώρηση μνήμης ώστε να μεγαλώσει η χωρητικότητα και να γίνουν οι απαραίτητες αντιγραφές (αυτά που κάνουμε με τη *renew()*). Βάζοντας αρκετά μεγάλη τιμή χωρητικότητας –με τη μέθοδο *reserve()* που ακολουθεί– περιορίζεις αυτές τις χρονοβόρες διαδικασίες.

- Η χωρητικότητα ενός αντικειμένου *vector* αλλάζει με τη

  ```
  void reserve( size_type n );
  ```

  Αν η *n* έχει τιμή μεγαλύτερη από την τρέχουσα χωρητικότητα (`a.capacity()`) τότε δεσμεύεται περισσότερη μνήμη ώστε να έχουμε `a.capacity()` ≥ *n*. Αυτό συνήθως απαιτεί αντιγραφή των περιεχομένων του *a*.

Επειδή με τις *size()*, *resize()*, *capacity()*, *reserve()* μπορεί να υπάρχουν μπερδεματα ας προσπαθήσουμε να τα ξεκαθαρίσουμε με ένα προγραμματάκι:

Παράδειγμα

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    vector<int> av( 3 );
 8:
 9:    cout << "capacity: " << av.capacity()
10:         << "   size: " << av.size() << endl;
11:    for ( int i(0); i <= 6; ++i )
12:    {
13:       if ( i < av.size() )  av[i] = i*10;
14:                        else  av.push_back( i*10 );
15:       cout << "i: " << i << "   capc: " << av.capacity()
16:            << "   size: " << av.size() << "   ";
17:       for ( int k(0); k < av.size(); ++k )
18:          cout << av[k] << ' ';
19:       cout << endl;
20:    }
21:
22:    av.reserve( 31 );
23:    av.resize( 4 );
24:    cout << "capacity: " << av.capacity()
25:         << "   size: " << av.size() << endl;
26:    for ( int k(0); k < av.size(); ++k )  cout << av[k] << ' ';
27:    cout << endl;
28:    try { cout << av.at(5) << endl; }
29:    catch( out_of_range& x )
30:    {  cout << x.what() << endl;  }
31: }
```

που μας δίνει:

```
capacity: 3   size: 3
i: 0   capc: 3   size: 3   0 0 0
i: 1   capc: 3   size: 3   0 10 0
i: 2   capc: 3   size: 3   0 10 20
i: 3   capc: 6   size: 4   0 10 20 30
i: 4   capc: 6   size: 5   0 10 20 30 40
i: 5   capc: 6   size: 6   0 10 20 30 40 50
i: 6   capc: 12   size: 7   0 10 20 30 40 50 60
capacity: 31   size: 4
0 10 20 30
index out of range in function: vector:: at(size_t)
 index: 5 is greater than max_index: 4
```

Με τη δήλωση της γραμμής 7 δηλώνουμε έναν (δυναμικό) πίνακα ακεραίων με αρχικό μέγεθος 3, δηλαδή με 3 στοιχεία. Από τη γραμμή 9 παίρνουμε το αποτέλεσμα "`capacity: 3  size: 3`". Η χωρητικότητα έγινε αυτομάτως `3`. Αυτή είναι η ελάχιστη τιμή για την οποίαν ικανοποιείται η: `av.size() ≤ av. capacity()`.

Στις πρώτες 3 εκτελέσεις (*i*: 0, 1, 2) της **for** δίνουμε τιμές στα `va[i]` με εκχώρηση (γραμμή 13) αφού τα στοιχεία αυτά υπάρχουν.

Για την εισαγωγή της 4ης τιμής ("`3`") χρησιμοποιούμε την "`av.push_back(i*10)`" διότι θέλουμε να μεγαλώσουμε ταυτοχρόνως τον πίνακα. Το ίδιο ισχύει και για τις επόμενες τιμές.

Όπως φαίνεται στη γραμμή "`i: 3`" η *av.size()* επέστρεψε τιμή "`4`" –αυξήθηκε κατά 1– αλλά η *av.capacity()* επέστρεψε "`6`"· διπλασιάστηκε με τη νέα παραχώρηση μνήμης που μεσολάβησε! Στη γραμμή "`i: 6`" βλέπουμε τι έγινε για να μπορέσουμε να βάλουμε και έβδομη τιμή: το μέγεθος αυξήθηκε κατά 1 αλλά η χωρητικότητα ξαναδιπλασιάστηκε· είχαμε και πάλι νέα παραχώρηση. Έτσι δουλεύει ο παραχωρητής μνήμης του *vector* και αυτό το έχουμε συζητήσει ήδη στην §16.13.3.

Η γραμμή "`capacity: 31  size: 4`" μας δείχνει τα αποτελέσματα των εντολών που δώσαμε στις γραμμές 22 και 23. Το αποτέλεσμα της **for** της γραμμής 26 μας επιβεβαιώνει το "`size: 4`".

Άλλη μια επιβεβαίωση για το μέγεθος μας δίνει και η απόπειρα χρήσης του 6ου στοιχείου του πίνακα με την "`av.at(5)`" μέσα σε μια `try`. Ρίχνεται εξαίρεση *out_of_range* από τη *what()* της οποίας παίρνουμε τις δύο τελευταίες γραμμές.

Τώρα θα πούμε και μερικά πράγματα για τους προσεγγιστές του *vector*. Αν ψάξεις λίγο στο αρχείο `vector` της C++ που χρησιμοποιείς θα δεις κάτι σαν:

```
template < class T, class Allocator = allocator<T> >
class vector
{
// . . .
public:
   typedef T           value_type;
// . . .
   typedef value_type* iterator;
// . . .
   typedef ptrdiff_t   difference_type;
// . . .
```

Δηλαδή: ο προσεγγιστής "`vector<T>::iterator`" είναι βέλος "`T*`" προς στοιχείο πίνακα. Έτσι, γίνεται αυτομάτως προσεγγιστής τυχαίας πρόσβασης και έχουμε δυνατότητα για αριθμητική βελών.

Στο παραπάνω παράδειγμα θα μπορούσαμε, μετά τη γραμμή 20, να βάλουμε:

```
   cout << *(av.begin()+3) << endl;
   vector<int>::iterator it1( av.begin() ),
                         it2( av.begin() );
   it1 += 2;
```

```
   it2 += 6;
   vector<int>::difference_type dist( it2 - it1 );
   cout << "dist: " << dist << endl;
```

και να πάρουμε:

```
30
*it1: 20   *it2: 60
dist: 4
```

Και κάτι για τις ακυρώσεις προσεγγιστών:

- ***Αν έχεις προσεγγιστές προς στοιχεία κάποιου αντικειμένου* vector *και κάνεις κάποια εισαγωγή (insert) ή διαγραφή (erase) όσοι προσεγγιστές έδειχναν από το πρώτο στοιχείο που εισάχθηκε (ή διαγράφηκε) και μετά ακυρώνονται (δεν δείχνουν αυτό που έδειχναν).***

Αν το ξεχάσεις μπορεί να «υποφέρεις» από μερικά δύσκολα προγραμματιστικά λάθη.

26.2.1.1 Εξαιρέσεις, Απόδοση και Άλλα

Με πόση ασφάλεια και πόσο γρήγορα γίνονται οι πράξεις σε αντικείμενα *vector*; Ας τις δούμε μια προς μια.

Πρώτα οι διαγραφές: Ας πούμε ότι έχεις ένα αντικείμενο *v* τύπου `vector <K>` –με *n* στοιχεία τύπου *K*– και δίνεις την εντολή "`v.erase(f, l);`".

Αν στα προς διαγραφή στοιχεία περιλαμβάνεται και το τελευταίο στοιχείο (αν δηλαδή *l* == *v*`.`*end()*) δεν γίνονται αντιγραφές. Απλώς ο παραχωρητής παίρνει πίσω τα τελευταία *l - f* στοιχεία και καταστρέφει τα αντικείμενά τους καλώντας τον `~K()`. Στην περίπτωση αυτή (καλώς εχόντων των πραγμάτων) δεν εγείρεται εξαίρεση. Ο χρόνος που απαιτείται είναι αυτός που χρειάζεται για την καταστροφή των *l - f* αντικειμένων.

Έστω τώρα ότι ο *f* δείχνει το `v[j]` και ο *l* δείχνει το `v[k]` (όπου *k - j = l - f*), δύο ενδιάμεσα στοιχεία του πίνακα. Τι πρέπει να γίνει;

- Να αντιγραφούν τα `v[k]..v[n-1]` στα στοιχεία που ξεκινούν από το `v[j]` (`copy(l, v.end(), f)`).
- Να καταστραφούν τα τελευταία *l - f* στοιχεία και να μειωθεί το μέγεθος του πίνακα κατά *l - f*.

Δηλαδή: έχουμε γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-1-*k*) των στοιχείων από το `v[k+1]` μέχρι το τέλος του πίνακα.

Από τις αντιγραφές μπορεί να έχουμε εξαίρεση. Πάντως δεν θα έχουμε διαρροή μνήμης και το *v* θα είναι διαχειρίσιμο (βασική εγγύηση).

Και για να επανέλθουμε σε αυτό που λέμε πιο πάνω: *αν έχεις προσεγγιστές που –πριν τη διαγραφή– δείχνουν σε στοιχεία από* `v[j]` *μέχρι το τέλος του πίνακα, μετά τη διαγραφή ακυρώνονται.*

Ας πάμε τώρα στην εισαγωγή. Έστω ότι θέλουμε να εισαγάγουμε *m* αντικείμενα ξεκινώντας από τη θέση `v[k]`. Θα πρέπει:

- Να μεγαλώσει το μέγεθος του πίνακα κατά *m* (σε *n*+*m*). Υποθέτουμε ότι δεν θα χρειαστεί νέα παραχώρηση μνήμης.
- Να αντιγραφούν τα `v[k]..v[n-1]` στα `v[k+m]..v[n+m-1]` αντιστοίχως.
- Να αντιγραφούν τα *m* εισαγόμενα αντικείμενα στις `v[k]..v[k+m-1]`.

Έχουμε δηλαδή και εδώ γραμμική εξάρτηση του χρόνου από το πλήθος (*n*-*k*) των στοιχείων από το `v[k]` μέχρι το τέλος του πίνακα και το πλήθος *m* των εισαγόμενων στοιχείων. Αλλά:

- Αν δεν ισχύει η υπόθεση που κάναμε και *έχουμε* νέα παραχώρηση μνήμης τότε θα πρέπει να γίνουν και *n* αντιγραφές. Στην περίπτωση αυτήν μπορεί –κατ' αρχήν τουλάχιστον– να πάρουμε *bad_alloc*.

- Ο χρόνος εισαγωγής στο τέλος εξαρτάται μόνο από το πλήθος των στοιχείων που εισάγονται. Φυσικά, για εισαγωγή ενός στοιχείου είτε με την *push_back()* είτε με κάποια *insert()* ο χρόνος είναι σταθερός. Σταθερός είναι ο χρόνος και την *pop_back()*.
- Για την τρίτη συνάρτηση (περίγραμμα) εισαγωγής

```
template < class InputIterator >
void insert( S<K>::iterator pos,
             InputIterator first, InputIterator last );
```

τα πράγματα μπορεί να είναι πιο πολύπλοκα. Οι *first* και *last* είναι –γενικώς– προσεγγιστές εισόδου. Και αν μεν είναι πρόσθιοι μπορούν να γίνουν αυτά που είπαμε παραπάνω. Αν όμως είναι, ας πούμε, *istream_iterator*s τότε δεν μπορούμε να μετρήσουμε και να βρούμε το *m* εκ των προτέρων. Τα στοιχεία θα εισάγονται ένα προς ένα και θα γίνουν *m* φορές αυτά που είπαμε πιο πάνω (αν τα εξειδικεύσεις για $m = 1$).

Εκτός από τη *bad_alloc*, που προαναφέραμε, μπορεί να έχουμε εξαίρεση από τις αντιγραφές. Σε κάθε περίπτωση έχουμε και εδώ ασφάλεια επιπέδου βασικής εγγύησης.

26.2.1.2 Και μια Εξειδίκευση: "vector<bool>"

Όπως έχουμε παρατηρήσει και σε προηγούμενο κεφάλαιο, από τα 8 ψηφία της ψηφιολέξης με την οποίαν παριστάνεται μια τιμή τύπου `bool` χρησιμοποιούμε μόνον το ένα. Για να γίνει η σχετική οικονομία, η STL προσφέρει μια εξειδίκευση του *vector* για τον τύπο `bool`:

```
template <class Allocator>
class vector<bool, Allocator>
```

Πέρα από την οικονομία μνήμης, η εξειδίκευση έχει δύο επιπλέον συναρτήσεις:

- Τη μέθοδο:

```
void flip();
```

που αλλάζει τις τιμές όλων των περιεχομένων από *b* σε *~b*· δηλαδή αλλάζει όλα τα `false` σε `true` και όλα τα `true` σε `false`.

- Πέρα από τη μέθοδο *swap()*, που ανταλλάσσει τις τιμές δύο πινάκων, τη στατική συνάρτηση:

```
static void swap( vector<bool>::reference ref1,
                  vector<bool>::reference ref2 );
```

που ανταλλάσσει τις τιμές δύο στοιχείων του πίνακα. Το "`reference`" να το σκέφτεσαι ως "`bool&`" (αλλά διάβασε παρακάτω.)

Παρατήρηση:▶

Το περίγραμμα *vector* έχει γενικώς έναν τύπο "`reference`" που για το περίγραμμα `vector<K>` ορίζεται ως "`K&`". Ειδικώς όμως για το `vector<bool>` υπάρχει πρόβλημα: Όπως έχουμε πει, οι αναφορές είναι «κρυμμένα» βέλη, δηλαδή διευθύνσεις. Αν λοιπόν «στριμώξουμε» το κάθε στοιχείο μιας τέτοιας κλάσης σε ένα δυαδικό ψηφίο τότε την ίδια διεύθυνση τη μοιράζονται 8 (ή περισσότερα) στοιχεία. Για να λυθεί αυτό το πρόβλημα ορίζεται (μέσα στο `vector<bool>`) ολόκληρη κλάση `reference`.◀

Δες το παρακάτω προγραμματάκι:

```
 0: #include <iostream>
 1: #include <vector>
 2: using namespace std;
 3:
 4: void display( ostream& tout, const vector<bool>& bv )
 5: {
 6:    tout << boolalpha;
 7:    for ( int k(0); k < bv.size(); ++k ) tout << bv[k] << ' ';
 8:    tout << endl;
 9: }
10:
11: int main ()
12: {
```

```
13:    vector<bool> bv1( 3 ) , bv2( 2 );
14:
15:    bv1[0] = true;  bv1[1] = false;  bv1[2] = true;
16:    cout << "bv1: ";  display( cout, bv1 );
17:    bv1.flip();
18:    cout << "bv1: ";  display( cout, bv1 );
19:
20:    bv2[0] = true;  bv2[1] = false;
21:    cout << "bv2: ";  display( cout, bv2 );
22:
23:    vector<bool>::swap( bv1[0], bv1[1] );
24:    cout << "bv1: ";  display( cout, bv1 );
25:    vector<bool>::swap( bv2.front(), bv2.back() );
26:    cout << "bv2: ";  display( cout, bv2 );
27:    vector<bool>::swap( bv1[1], bv2[1] );
28:    cout << "bv1: ";  display( cout, bv1 );
29:    cout << "bv2: ";  display( cout, bv2 );
30:
31:    bv1.swap( bv2 );
32:    cout << "bv1: ";  display( cout, bv1 );
33:    cout << "bv2: ";  display( cout, bv2 );
34: }
```

που βγάζει:

```
bv1: true false true
bv1: false true false
bv2: true false
bv1: true false false
bv2: false true
bv1: true true false
bv2: false false
bv1: false false
bv2: true true false
```

Πρόσεξε τα εξής:

- Η αλλαγή του *bv1* από την πρώτη γραμμή στη δεύτερη προκλήθηκε από την "`bv1.flip();`" της γραμμής 17.
- Η αλλαγή του *bv1* από τη δεύτερη στην τέταρτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 23 ενώ η αλλαγή του *bv2* από τη τρίτη στην πέμπτη γραμμή είναι αποτέλεσμα της *swap()* της γραμμής 25.
- Η *swap()* της γραμμής 27 ανταλλάσσει τις τιμές στοιχείων διαφορετικών αντικειμένων.
- Τέλος, στη γραμμή 31 βλέπεις την «άλλη *swap()*», αυτήν που έχουν όλα τα περιέχοντα της STL. Το αποτέλεσμα φαίνεται στις δύο τελευταίες του απότελέσματος: τα *bv1* και *bv2* έχουν ανταλλάξει τιμές. Το ίδιο αποτέλεσμα θα είχαμε με την "`bv2.swap(bv1);`".

26.2.2 Το Περίγραμμα "*deque*"

Το περίγραμμα *deque* (προφέρεται "deck") υλοποιεί αυτό που λέει το όνομά της: μια ουρά δύο άκρων (*d*ouble *e*nded *que*ue).

Για να μπορεί να λειτουργήσει ως ουρά δύο άκρων χρειάζεται –εκτός από τις *front()*, *back()*, *push_back()*, *pop_back()*– και τις *push_front()*, *pop_front()* για εισαγωγή και διαγραφή στην αρχή του περιέχοντος που τις έχουμε δει στο *SListT* (§25.7.1):

- Εισαγωγή στοιχείου στην αρχή του περιέχοντος:
  `void deque<K>::push_front( const K& val );`
  Μετά την "`a.push_front( c );`" η "`a.front()`" μας δίνει το *c*· το ίδιο και η "`*a.begin()`".
- Διαγραφή του πρώτου στοιχείου της ακολουθίας του περιέχοντος (αν δεν είναι κενό):
  `void deque<K>::pop_front();`

Οι δύο αυτές μέθοδοι έχουν σταθερό χρόνο εκτέλεσης, όπως άλλωστε και οι *push_back()*, *pop_back()*.

Κατά τα άλλα, παρ' όλο που ο τρόπος αποθήκευσης των περιεχομένων είναι πιο πολύπλοκος από τον απλό μονοδιάστατο πίνακα, η λειτουργικότητα του *deque* είναι ίδια με αυτήν του *vector* με μια διαφορά: *Το deque δεν έχει τις reserve() και capacity()*.

26.2.3 Το Περίγραμμα "*list*"

Το περίγραμμα *list*:

```
template < class T, class Allocator = allocator<T> >
class list;
```

υλοποιεί λίστα με διπλή σύνδεση με περιεχόμενο στοιχείων τύπου *T*.

Για το *list*, πέρα από αυτά που έχει το *deque*, υπάρχουν επιπλέον:

- Μέθοδοι για μεταφορά στοιχείων (προσοχή: *μεταφορά όχι αντιγραφή*) σε μια λίστα από άλλη λίστα με στοιχεία και παραχωρητή μνήμης ίδιου τύπου:

  ```
  void splice( iterator pos, list<T,Allocator>& x );
  void splice( iterator pos, list<T,Allocator>& x, iterator i);
  void splice( iterator pos, list<T,Allocator>& x,
               iterator first, iterator last );
  ```

  Αν οι *l1*, *l2* είναι λίστες `list<K>` τότε:

  - Η "`l1.splice(it, l2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1*, μεταφέρει στην *l1* όλα τα στοιχεία της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it*. Η *l2* παραμένει χωρίς στοιχεία.
  - Η "`l1.splice(it, l2, it2);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και ο *it2* προσεγγιστής προς στοιχείο της *l2*, μεταφέρει στην *l1* το στοιχείο που δείχνει ο *it2* και το συνδέει πριν από αυτό που δείχνει ο *it* ενώ το αφαιρεί από την *l2*. Αν δώσεις "`l1.splice(it, l1, it2);`" κάνεις εσωτερική μετακίνηση στην *l1*.
  - Η "`l1.splice(it, l2, f, l);`", όπου ο *it* είναι προσεγγιστής προς στοιχείο της *l1* και οι *f*, *l* προσεγγιστές προς στοιχεία της *l2*, μεταφέρει στην *l1* το στοιχεία της περιοχής `[f,l)` της *l2* και τα συνδέει πριν από αυτό που δείχνει ο *it* ενώ τα αφαιρεί από την *l2*. Αν δώσεις "`l1. splice(it, l1, f, l);`" έχεις εσωτερική μετακίνηση στοιχείων στην *l1*. Στην περίπτωση αυτήν ο *it* πρέπει να είναι έξω από την `[f,l)`.

- Μέθοδος για διαγραφή στοιχείων με συγκεκριμένη τιμή και μέλος–περίγραμμα για διαγραφή στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void remove( const T& v );
  template < class Predicate >
  void remove_if( Predicate pred );
  ```

  Η "`l3.remove( v );`" έχει ως αποτέλεσμα την αφαίρεση από την *l3* όλων των στοιχείων που έχουν τιμή *v*.

  Μέσω του *pred()*, που είναι ένα κατηγόρημα με μια παράμετρο τύπου *T*, περνάμε στη *remove_if()* τις προδιαγραφές. Δες ένα

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
 6: {
 7:    for ( list<K>::iterator it(lv.begin());
```

```
 8:          it != lv.end(); ++it ) tout << *it << ' ';
 9:    tout << endl;
10: } // display
11:
12: struct Dec0
13: {
14:    bool operator()( const int& v )
15:    {  return ( (0 <= v) && (v < 10) );  }
16: }; // Dec0
17:
18: int main ()
19: {
20:    int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
21:    list<int> l3( ints, ints+11 );
22:    display<int>( cout, l3 );
23:
24:    l3.remove( 23 );  display<int>( cout, l3 );
25:
26:    l3.remove( 7 );  display<int>( cout, l3 );
27:
28:    l3.remove_if( Dec0() );  display<int>( cout, l3 );
29: }
```

δίνει:

```
3 23 3 19 5 19 17 7 7 17 23
3 3 19 5 19 17 7 7 17
3 3 19 5 19 17 17
19 19 17 17
```

Με την εντολή της γραμμής 24 ζητούμε να αφαιρεθούν από τη λίστα όλα τα "**23**" και παίρνουμε τη δεύτερη γραμμή του αποτελέσματος. Παρομοίως, με την εντολή της γραμμής 26 αφαιρούμε τα "**7**" και παίρνουμε την τρίτη γραμμή του αποτελέσματος.

Στις γραμμές 12-16 ορίζουμε ένα συναρτησοειδές (*Dec0*) που επιλέγει μονοψήφιους φυσικούς. Με την εντολή της γραμμής 28 αφαιρούμε από τη λίστα τους μονοψήφιους φυσικούς "**3**", "**3**" και "**5**".

Αντί για το συναρτησοειδές θα μπορούσαμε να ορίσουμε:

```
bool dec0( int v )  {  return ( (0 <= v) && (v < 10) );  }
```

και να πάρουμε το ίδιο αποτέλεσμα γράφοντας:

```
   l3.remove_if( dec0() );
```

❖❖❖

- Μέθοδος για διαγραφή *διαδοχικών αντιγράφων ενός στοιχείου* και μέλος–περίγραμμα για διαγραφή διαδοχικών στοιχείων με συγκεκριμένες προδιαγραφές:

  ```
  void unique();
  template < class BinaryPredicate >
  void unique( BinaryPredicate binaryPred );
  ```

  Για την πρώτη περίπτωση διασχίζεται η λίστα με έναν προσεγγιστή *it* και συγκρίνονται για ισότητα ζεύγη περιεχόμενων τιμών (***it**, ***(it-1)**). Αν βρεθούν ίσα διαγράφεται αυτό που δείχνει ο *it*.

  Για τη δεύτερη μορφή, το κατηγόρημα «τροφοδοτείται» με τα ***it**, ***(it-1)**. Αν το κατηγόρημα επιστρέψει τιμή **true** διαγράφεται το στοιχείο που δείχνει ο *it*. Όπως καταλαβαίνεις, το κατηγόρημα πρέπει να είναι δυαδικό.

Παράδειγμα

Το πρόγραμμα:

```
 0: #include <iostream>
 1: #include <list>
 2: using namespace std;
 3:
 4: template< typename K >
 5: void display( ostream& tout, list<K>& lv )
```

```
 6: // ΟΠΩΣ ΠΑΡΑΠΑΝΩ
 7:
 8: struct NotInOrder
 9: {
10:    bool operator()( const int& v1, const int& v2 )
11:    {  return ( v1 > v2 );  }
12: }; // NotInOrder
13:
14: int main ()
15: {
16:    int ints[] = { 2, 2, 2, 3, 3, 5, 5, 5, 5, 2, 2, 3 };
17:    list<int> l4( ints, ints+12 );
18:    display<int>( cout, l4 );
19:
20:    l4.unique();  display<int>( cout, l4 );
21:
22:    l4.unique( NotInOrder() );  display<int>( cout, l4 );
23: }
```

δίνει:

```
2 2 2 3 3 5 5 5 5 2 2 3
2 3 5 2 3
2 3 5
```

Τι γίνεται με την εντολή της γραμμής 20; Μετά το πρώτο "2" διαγράφονται τα δύο αντίγραφά του ("2") που το ακολουθούν· τα ίδια ισχύουν και για τα "3" και τα "5". Το "2", που είναι προτελευταίο στοιχείο της ακολουθίας, αφαιρείται διότι υπάρχει "2" ακριβώς πριν από αυτό.

Το συναρτησοειδές *NotInOrder* ορίζει ένα κατηγόρημα που επιστρέφει `true` αν το πρώτο όρισμα είναι μεγαλύτερο από το δεύτερο. Η εντολή της γραμμής 24 έχει ως αποτέλεσμα να αφαιρεθεί το "2" που ακολουθεί το "5" και στη συνέχεια, να αφαιρεθεί και το "3" όταν βρίσκεται και αυτό να ακολουθεί το "5".

- Μέθοδος και μέλος–περίγραμμα για ταξινόμηση των περιεχομένων της λίστας:

  ```
  void sort();
  template < class Compare >
  void sort( Compare comp );
  ```

  Τα περιεχόμενα μιας λίστας τύπου `list<K>` ταξινομούνται με βάση τον τελεστή "<" της *K*.

  Αν θέλουμε ταξινόμηση χωρίς να έχουμε ορισμένο τον "<" της *K* ή με άλλη διάταξη από αυτήν που μας δίνει ο γράφουμε ένα συναρτησοειδές-κατηγόρημα που να ορίζει τη διάταξη ("<" της *K*) που μας ενδιαφέρει. Για την ταξινόμηση η "*a* < *b*" έχει το νόημα «το *a* προηγείται του *b*».

Παράδειγμα

Σε μια λίστα έχουμε τις ημερομηνίες ιστορικών γεγονότων:

```
Date dates[] = {  Date(1821,3,25), Date(1940,10,28),
                  Date(1904,10,13), Date(1453,5,29),
                  Date(1912,10,26)  };
list<Date> dateList( dates, dates+5 );
```

Αν δώσουμε:

```
dateList.sort();
display<Date>( cout, dateList );
```

θα πάρουμε:

```
29.5.1453 25.3.1821 13.10.1904 26.10.1912 28.10.1940
```

Όπως βλέπεις, εδώ έχουμε ιστορικές ημερομηνίες τις επετείους των οποίων γιορτάζουμε πανελληνίως ή τοπικώς. Θα ήταν λοιπόν ενδιαφέρον να τις έχουμε ταξινομημένες κατά μήνα και ημέρα. Για να το κάνουμε αυτό γράφουμε το συναρτησοειδές:

```
struct MonthFirst
{
   bool operator()( const Date& d1, const Date& d2 )
   {
      bool fv( false );
      if ( d1.getMonth() < d2.getMonth() )  fv = true;
      else if ( d1.getMonth() == d2.getMonth() )
      {
         if ( d1.getDay() < d2.getDay() )  fv = true;
         else if ( d1.getDay() == d2.getDay() )
            fv = ( d1.getYear() < d2.getYear() );
      }
      return fv;
   }
}; // MonthFirst
```

και γράφοντας:

```
   dateList.sort( MonthFirst() );
   display<Date>( cout, dateList );
```

θα πάρουμε:

```
25.3.1821 29.5.1453 13.10.1904 26.10.1912 28.10.1940
```

Αν, τώρα, θέλουμε τις ημερομηνίες από τις πιο πρόσφατες προς τις παλαιότερες (κατά «φθίνουσα» τάξη) γράφουμε το συναρτησοειδές:

```
struct Decr
{
   bool operator()( const Date& d1, const Date& d2 )
   {  return ( d2 < d1 );  }
}; // Decr
```

και με τις:

```
   dateList.sort( Decr() );
   display<Date>( cout, dateList );
```

να πάρουμε:

```
28.10.1940 26.10.1912 13.10.1904 25.3.1821 29.5.1453
```

❖❖❖

Γιατί να προτιμήσουμε να δώσουμε "`l.sort()`" αντί για "`sort(l.begin(), l.end())`"; Η μέθοδος υπερέχει ως προς το ότι είναι **ευσταθής** (stable): η σχετική διάταξη δύο αντικειμένων με ίσα κλειδιά δεν αλλάζει κατά την ταξινόμηση. Προφανώς αυτό έχει νόημα όταν η λίστα φιλοξενεί σύνθετα αντικείμενα.

- Μέθοδος και μέλος–περίγραμμα για να συγχωνεύσουμε (τα περιεχόμενα από) δύο λίστες που είναι ήδη ταξινομημένες:

  ```
  void merge( list& x );
  template < class Compare >
  void merge( list& x, Compare comp );
  ```

  Αν έχουμε "`list<K> l1, l2`" και οι *l1*, *l2* είναι ταξινομημένες σύμφωνα με τον "`<`" της *K* τότε η "`l1.merge(l2);`" μεταφέρει (δεν αντιγράφει) και συγχωνεύει στην *l1* τα περιεχόμενα της *l2*. Τελικώς:

  - Η *l1* έχει όλα τα στοιχεία που είχαν αρχικώς οι *l1*, *l2* και είναι ταξινομημενα σύμφωνα με τον "`<`" της *K*.
  - Η *l2* είναι *κενή*.

  Αν οι *l1*, *l2* είναι ταξινομημένες με άλλη λογική χρησιμοποιούμε το περίγραμμα βάζοντας με τη δεύτερη παράμετρο τη διάταξη όπως κάναμε στην ταξινόμηση.

  Σε σχέση με τον αλγόριθμο (*std*::)*merge()* οι μέθοδοι έχουν ευστάθεια με την έννοια: μετά την "`l1.merge(l2);`" κάθε στοιχείο της *l2* εισάγεται μετά από το(τα) στοχείο(-α) της *l1* με το ίδιο κλειδί.

- Μέθοδος για την αντιστροφή μιας λίστας:

```
void reverse();
```

Για παράδειγμα, οι εντολές:

```
int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
list<int> l3( ints, ints+11 );
display<int>( cout, l3 );

l3.reverse();  display<int>( cout, l3 );
```

θα δώσουν:

```
3 23 3 19 5 19 17 7 7 17 23
23 17 7 7 17 19 5 19 3 23 3
```

26.2.4 Ποια Ακολουθία να Διαλέξω;

Η πιο συνηθισμένη απάντηση είναι: *vector*! Πράγματι, αυτό το περιέχον φαίνεται να υπερέχει σε όλες τις περιπτώσεις αλλά και τα άλλα έχουν λόγο ύπαρξης.

- Διάλεξε το *deque* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στα δύο άκρα.
- Διάλεξε το *list* αν έχεις συλλογή-ακολουθία με συχνές εισαγωγές/εξαγωγές στη «μέση» (μακριά από τα άκρα).

26.3 Συνειρμικά Περιέχοντα

Συνειρμική μνήμη (associative memory) ή **μνήμη προσπελάσιμη με το περιεχόμενο** (content-addressable memory) είναι τρόπος οργάνωσης της μνήμης ώστε η πρόσβαση στην κάθε μονάδα του περιεχομένου της να γίνεται όχι με τη διεύθυνσή της αλλά με τμήμα της μονάδας (κλειδί). Έτσι οργανώνουν τα περιεχόμενά τους τα **συνειρμικά περιέχοντα** (associative containers) της STL.

Δες τις επικεφαλίδες των τεσσάρων περιγραμμάτων:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Η πρώτη παράμετρος-κλάση (*Key*) είναι ο τύπος του αντικειμένου-κλειδιού με βάση το οποίο γίνονται οι αναζητήσεις στη συλλογή. Η παράμετρος-κλάση *Compare* είναι ένα συναρτησοειδές που καθορίζει διάταξη "`<`" για αντικείμενα κλάσης *Key*. Αν για την *Key* είναι ορισμένος ο "`<`" μπορείς να μην ορίσεις το συναρτησοειδές και τον ρόλο του να παίξει το *less*`<`*Key*`>`.[9]

Για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "`<`" που είδαμε πιο πάνω– σε ένα **δένδρο δυαδικής αναζήτησης** (binary search tree).[10] Όπως

[9] Είδαμε το περίγραμμα "*less*" στην §25.6.3.

[10] Συνήθως ένα δένδρο red-black.

κατάλαβαίνεις, παρ' όλο που πραγματευόμαστε –γενικώς– σύνολα και πολυσύνολα, συζητούμε για **διαταγμένα** (ordered) σύνολα και πολυσύνολα.[11]

Κάθε στιγμιότυπο οποιουδήποτε από τα παραπάνω περιγράμματα έχει τη δική του κλάση (αμφίδρομων) προσεγγιστών.

Πέρα από τις μεθόδους που έχουν όλα τα περιέχοντα της STL, στα συνειρμικά περιέχοντα υπάρχουν:

- Μέθοδος που επιστρέφει το πλήθος των αντικειμένων που έχουν συγκεκριμενο κλειδί.
  ```
  size_type A<K>::count( const K& x ) const;
  ```
  Προφανώς, αν το *A* είναι *set* ή *map* η *count()* μπορεί να επιστρέφει "`0`" ή "`1`".[12]
- Μέθοδοι για εντοπισμό κλειδιού στη συλλογή:
  ```
  A<K>::iterator A<K>::find( const K& x );
  A<K>::const_iterator A<K>::find( const K& x ) const;
  ```
  Αν το κλειδί *x* βρεθεί στη συλλογή επιστρέφεται προσεγγιστής προς αυτό το αντικείμενο. Αλλιώς ο προσεγγιστής δείχνει το "`A<K>::end()`".
- Μέθοδοι για εισαγωγές στοιχείων που θα τις δούμε ξεχωριστά για κάθε περιέχον.
- Μέθοδοι για διαγραφές στοιχείων:
  ```
  void A<K>::erase( A<K>::iterator pos );
  size_type A<K>::erase( const K& x );
  void A<K>::erase( A<K>::iterator first, A<K>::iterator last );
  void A<K>::clear();
  ```
  Η πρώτη διαγράφει το αντικείμενο που δείχνει ο *pos*.

  Η δεύτερη διαγράφει όλα τα αντικείμενα που έχουν κλειδί *x*. Επιστρέφει το πλήθος των αντικειμένων που διαγράφηκαν· το ίδιο που θα επέστρεφε το "`count(x)`" πριν από τη διαγραφή.

  Η τρίτη διαγράφει όλα τα αντικείμενα που τα κλειδιά τους ανήκουν στην περιοχή [*first*, *last*).

  Η τελευταία είναι ισοδύναμη με "`erase( begin(), end() )`" και έχει ως αποτέλεσμα κενό περιέχον.
- Μέθοδοι για εντοπισμό περιοχής κλειδιών με συγκεκριμένη τιμή:
  ```
  A<K>::iterator A<K>::lower_bound( const K& x );
  A<K>::const_iterator A<K>::lower_bound( const K& x ) const;
  ```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο που δεν είναι μικρότερο από *x*. Δες και αυτά που λέγαμε για τη συνάρτηση *lower_bound()* στην §26.1.3. Αν στο περιέχον υπάρχει (-ουν) στοιχείο(-α) με κλειδί *x* τότε ο προσεγγιστής που επιστρέφεται δείχνει το πρώτο (ή το μοναδικό) στοιχείο με τέτοιο κλειδί.
  ```
  A<K>::iterator A<K>::upper_bound( const K& x );
  A<K>::const_iterator A<K>::upper_bound( const K& x ) const;
  ```
  Επιστρέφει προσεγγιστή προς το πρώτο στοιχείο του περιέχοντος που είναι μεγαλύτερο από *x*.
  ```
  pair< A<K>::iterator, A<K>::iterator > A<K>::equal_range( const K& x );
  pair< A<K>::const_iterator, A<K>::const_iterator >
                                    A<K>::equal_range( const K& x ) const;
  ```
  Επιστρέφεται ζεύγος[13] προσεγγιστών *f*, *l* ώστε όλα τα κλειδιά στην [*f*, *l*) να έχουν τιμή *x*. Αν δεν υπάρχει το κλειδί οι δυο προσεγγιστές είναι ίσοι (περιοχή κενή). Αν για ένα

---

[11] Το νέο πρότυπο C++11 προδιαγράφει και *μη-διατεταγμένα* σύνολα/πολυσύνολα.

[12] Γράφουμε "`A<K>`" αλλά μπορεί να είναι και "`A<K,C>`" ή "`A<K,C,T>`". Απλώς όλες οι μέθοδοι έχουν να κάνουν μόνο με το κλειδί.

συγκεκριμένο περιέχον πάρουμε το *lb* (*lower_bound()*), το *ub* (*upper_bound()*) και το ζεύγος *eqr* (*equal_range()*) τότε θα έχουμε *eqr*.*first* == *lb* και *eqr*.*second* == *ub*. Το πρόγραμμα:

```
#include <iostream>
#include <functional>
#include <set>
using namespace std;
int main()
{
   int ints[] = { 3, 23, 3, 19, 5, 19, 17, 7, 7, 17, 23 };
   multiset<unsigned int> ms( ints, ints+11 );
   typedef multiset<unsigned int>::iterator msIterator;

   msIterator lb( ms.lower_bound(17) );
   msIterator ub( ms.upper_bound(17) );
   pair<msIterator, msIterator> eqr( ms.equal_range(17) );
   if ( lb == eqr.first )  cout << "lb == eqr.first" << endl;
   if ( ub == eqr.second )  cout << "ub == eqr.second" << endl;
}
```

δίνει

```
lb == eqr.first
ub == eqr.second
```

- Μέθοδος που δίνει αντίγραφο του αντικείμενου-συναρτησοειδούς που χρησιμοποιεί η κλάση:

  `C A<K,C>::key_comp() const;`

  Για τη μέθοδο αυτή θα συζητήσουμε στη συνέχεια.

Κάθε ένα από τα περιγράμματα έχει τη δική του κλάση *αμφίδρομων* προσεγγιστών για τους οποίους ισχύουν τα εξής:

♦ ***Οι πράξεις διαγραφής δεν ακυρώνουν προσεγγιστές εκτός από αυτούς που δείχνουν στοιχεία που διαγράφονται.***

Στη συνέχεια και για κάθε περίγραμμα θα δούμε τις σχετικές πράξεις εισαγωγής (*insert()*). Αλλά από τώρα θα τονίσουμε ότι:

♦ ***Οι πράξεις εισαγωγής δεν ακυρώνουν προσεγγιστές.***

26.3.1 Το Περίγραμμα "*set*"

Το περίγραμμα "*set*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class set;
```

«φιλοξενεί» συλλογές αντικειμένων που είναι **σύνολα** (sets). Αυτό σημαίνει ότι σε κάθε συλλογή το κάθε αντικείμενο είναι *μοναδικό*.

Δεν υπάρχουν μέθοδοι για το *set* πέρα από αυτές που είδαμε για τα συνειρμικά περιέχοντα γενικώς.

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις στο πρόγραμμά σου:

```
#include <set>
```

Με τη:

```
set< T >  s;
```

δηλώνουμε το *s* ως σύνολο με στοιχεία τύπου *T*. Η τιμή εκκίνησης του *s* είναι το ∅. Π.χ. οι

```
   set< int > intSet;
// intSet == { }
   cout << "#intSet == " << intSet.size() << endl;
```

[13] Το περίγραμμα (*std*::)*pair* είναι ίδιο με το *PairT* που είδαμε στην §25.3.1.

```
   if ( intSet.empty() )  cout<< "intSet == {}" << endl;
                     else  cout<< "intSet != {}" << endl;
```

θα δώσουν:

```
#intSet == 0
intSet == {}
```

- Το "`0`" είναι ο **πληθάριθμος** (cardinality) του *intSet* και μας το δίνει η *intSet*`.`*size*`()`.
- Η *intSet*`.`*empty*`()` δίνει `true` που σημαίνει «Ναι, το *intSet* είναι κενό.»

  Με την *intSet*`.`*clear*`()` δίνεις τιμή ∅ στο *intSet*.

Μπορείς να δώσεις ως αρχική τιμή μια υποπεριοχή είτε από συνήθη πίνακα είτε από κάποιο άλλο περιέχον της STL:

```
   int            intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
   vector< int >  intVec( intArr+3, intArr+7 );
// . . .
   set< int > intSet1( intVec.begin(), intVec.end() );
   copy( intSet1.begin(), intSet1.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
   set< int > intSet2( intArr, intArr+5 );
   copy( intSet2.begin(), intSet2.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
-11 8 31 41
-23 -11 8 34 72
```

Πρόσεξε ότι τα στοιχεία των συνόλων μας δίνονται ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Στο θέμα αυτό θα επανέλθουμε στη συνέχεια.

Ο έλεγχος για να δούμε αν μια τιμή x ανήκει σε ένα σύνολο s ($x \in s$) μπορεί να γίνει με τρεις τρόπους:

- Με τη μέθοδο *lower_bound()*. Η "`*(s.lower_bound(x)) == x`" θα επιστρέψει `true` αν η τιμή x υπάρχει στο s. Οι

```
   if ( *(intSet1.lower_bound(31)) == 31 )
      cout << "31 in intSet1" << endl;
   else
      cout << "31 not in intSet1" << endl;
   if ( *(intSet1.lower_bound(61)) == 61 )
      cout << "61 in intSet1" << endl;
   else
      cout << "61 not in intSet1" << endl;
```

  θα δώσουν:

```
31 in intSet1
61 not in intSet1
```

- Με τη μέθοδο *find()*. Η "`s.find(x)`" θα επιστρέψει "`s.end()`" αν η τιμή x δεν υπάρχει στο s. Αλλιώς θα επιστρέψει προσεγγιστή *it*, τύπου `set<int> ::iterator` (το "`int`" για το παράδειγμά μας), τέτοιον ώστε `*`*it* `==` x. Τα ίδια αποτελέσματα με αυτά που πήραμε παραπάνω θα δώσουν και οι

```
   if ( intSet1.find(31)==intSet1.end() )
      cout << "31 not in intSet1" << endl;
   else
      cout << "31 in intSet1" << endl;
   if ( intSet1.find(61)==intSet1.end() )
      cout << "61 not in intSet1" << endl;
   else
      cout << "61 in intSet1" << endl;
```

- Με τη μέθοδο *count()*. Η "`s.count(x)`" θα επιστρέψει "`0`" αν η τιμή x δεν υπάρχει στο s· αλλιώς θα επιστρέψει "`1`". Οι

```
   if ( intSet1.count(31)==0 )  cout << "31 not in intSet1" << endl;
```

```
                          else  cout << "31 in intSet1" << endl;
   if ( intSet1.count(61)==0 )  cout << "61 not in intSet1" << endl;
                          else  cout << "61 in intSet1" << endl;
```

θα δώσουν τα ίδια αποτελέσματα.

Αν θέλεις μπορείς να δοκιμάσεις τα παραπάνω και με το στιγμιότυπο "`set<Date>`"· χρησιμοποίησε τον πίνακα "`dates`" που είδαμε στο παράδειγμα για την ταξινόμηση περιεχομένων λίστας.

Για να χειρίζεσαι ένα σύνολο έχεις ακόμη:

- Τις μεθόδους *erase()* –που είδαμε για τα συνειρμικά περιέχοντα– και επιτρέπουν τη διαγραφή στοιχείων από σύνολο
- Μεθόδους *insert()* που επιτρέπουν εισαγωγή στοιχείων σε σύνολο κλάσης `set<K>`:

```
pair< set<K>::iterator, bool > insert( const K& x );
set<K>::iterator insert( set<K>::iterator position, const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Η πρώτη μορφή επιστρέφει ζεύγος (*pair*) του οποίου

- Το μέλος *first* είναι τύπου `set<K>::iterator` και δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.
- Το μέλος *second* είναι τύπου `bool` και έχει τιμή `true` αν έγινε η εισαγωγή ή`false` αν δεν έγινε εισαγωγή διότι η τιμή υπήρχε.

Η δεύτερη μορφή είναι λίγο «περίεργη»: στην "`insert(it,v)`" με την πρώτη παράμετρο *it* κάνουμε μια υπόδειξη για το *πού* πρέπει να γίνει η εισαγωγή της *v*! Βεβαίως η εισαγωγή θα γίνει εκεί που πρέπει να γίνει αλλά αν ο *it* δείχνει *ακριβώς πριν*[14] από τη θέση που θα γίνει η εισαγωγή κερδίζουμε χρόνο. Επιστρέφει τιμή τύπου `set<K>::iterator` που δείχνει το μέλος στο οποίο βρίσκεται η τιμή *x*.

Με την τρίτη μορφή εισάγουμε στο σύνολο όλες τις τιμές μιας περιοχής [*first*, *last*).

Από τις εντολές:

```
   set<int> intSet;
   pair< set<int>::iterator, bool > retVal( intSet.insert(17) );
   cout << boolalpha
        << *(retVal.first) << "  " << retVal.second << endl;
```

παίρνουμε:

```
17  true
```

Ο προσεγγιστής `retVal.first` δείχνει το στοιχείο του *intSet* που έχει τιμή "`17`". Η τιμή `true` του `retVal.second` μας λέει ότι η εισαγωγή αυτής της τιμής έγινε τώρα.

Αν

```
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
```

από τις εντολές:

```
   intSet.insert( intArr, intArr+4 );
   copy( intSet.begin(), intSet.end(),
                                ostream_iterator<int>(cout, " ") );
   cout << endl;
```

παίρνουμε:

```
-23 8 17 34 72
```

Μετά από αυτές δίνουμε:

```
   retVal = intSet.insert( 8 );
   cout << *(retVal.first) << "  " << retVal.second << endl;
```

και παίρνουμε:

---

[14] Με το νέο πρότυπο C++11 έχουμε βέλτιστη ταχύτητα αν ο *it* δείχνει *ακριβώς μετά* τη θέση που θα γίνει η εισαγωγή.

```
8  false
```

Και εδώ ο `retVal.first` δείχνει στοιχείο με την τιμή που εισάγουμε αλλά η `retVal.second` έχει τιμή `false`: το "`8`" υπάρχει ήδη στο *intSet*.

Συνεχίζουμε δίνοντας:

```
set<int>::iterator it;
it = intSet.insert( retVal.first, 11 );
cout << *it << endl;
```

και παίρνουμε:

```
11
```

Δηλαδή: χρησιμοποιώντας τη δεύτερη μορφή της *insert()* εισάγουμε την τιμή "`11`". Ως πρώτο όρισμα βάζουμε τον προσεγγιστή που πήραμε από την εισαγωγή του "`8`". Η αποπαραπομπή του προσεγγιστή που μας επιστρέφεται μας δίνει ακριβώς το "`11`".

Ε, τώρα ας κάνουμε και μερικές διαγραφές:

```
int n( intSet.erase(72) );
cout << n << endl;
copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;
intSet.erase( intSet.begin(), intSet.find(17) );
copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;
```

που δίνουν:

```
1
-23 8 11 17 34
17 34
```

Η "`intSet.erase(72)`" επιστρέφει (και κάνει τιμή της *n*) το πλήθος των στοιχείων που διαγράφηκαν. Το "`1`" της πρώτης γραμμής μας λέει ότι διαγράφηκε ένα στοιχείο· δηλαδή το "`72`" υπήρχε και διαγράφηκε.

Η επόμενη «μαζική» διαγραφή διαγράφει τα στοιχεία του συνόλου από την αρχή μέχρι να βρει το "`17`".

Τέλος, να επισημάνουμε κάτι για τους προσεγγιστές του *set*: Είτε γράψεις "`set<K>::iterator it`" είτε γράψεις "`set<K>::const_iterator it`" *δεν επιτρέπεται να τροποποιήσεις το* `*`*it*. Γιατί; Διότι *τα περιεχόμενα ενός set είναι κλειδιά και* –όπως έχουμε τονίσει και σε προηγούμενα κεφάλαια– *τα κλειδιά δεν τροποποιούνται*. Αν επιτρεπόταν η τροποποίηση του `*`*it* θα ήταν πολύ απλό να καταστρέψεις και τη μοναδικότητα των στοιχείων του συνόλου και τη διάταξή τους.[15] Ο «νόμιμος» τρόπος να τροποποιήσεις το `*`*it* είναι ο εξής:

```
   K temp( *it );
   oneSet.erase( it );
// άλλαξε την τιμή της temp
   oneSet( temp );
```

Αν ο *K* είναι τύπος με μεγάλα αντικείμενα αυτός ο τρόπος γίνεται χρονοβόρος. Για την περίπτωση αυτήν υπάρχει άλλο περιέχον για παράσταση συνόλου: το "*map*" που θα δούμε στη συνέχεια.

26.3.1.1 Σχέσεις και Πράξεις Συνόλων

Η σχέση "⊆" («είναι υποσύνολο», «περιλαμβάνεται») καθώς και οι πράξεις "∪" (ένωση), "∩" (τομή), "\" (διαφορά) και "Δ" (συμμετρική διαφορά) υλοποιούνται με περιγράμματα συναρτήσεων. Για να τα χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <algorithm>`".

---

[15] Ο μεταγλωττιστής Borland C++ v.5.5 *επιτρέπει* τροποποίηση του `*`*it*! Αν τον έχεις δοκίμασε να κάνεις τις «παρανομίες» που λέμε.

Η σχέση $A \subseteq B$ (διαβάζεται ως «B includes A») υλοποιείται με το περίγραμμα *includes()* που ορίζεται στο **algorithm**:

```
template< class InputIter1, class InputIter2 >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2 );
template< class InputIter1, class InputIter2,
          class Compare >
bool includes( InputIter1 first1, InputIter1 last1,
               InputIter2 first2, InputIter2 last2,
               Compare comp);
```

Με τις:

```
#include <set>
#include <algorithm>
// . . .
   int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };
// . . .
   set<int> intSet1( intArr, intArr+3 );
   copy( intSet1.begin(), intSet1.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   set<int> intSet2( intArr, intArr+7 );
   copy( intSet2.begin(), intSet2.end(),
         ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intSet1.begin(), intSet1.end()) )
      cout << "intSet2 includes intSet1" << endl;
   else
      cout << "intSet2 does not include intSet1" << endl;
```

θα πάρουμε:

```
-23 34 72
-23 -11 8 31 34 41 72
intSet2 includes intSet1
```

Η σχέση $A \subset B$ –**γνήσιο υποσύνολο** (proper ή strict subset)– δεν υλοποιείται απ' ευθείας· θα πρέπει να τη δεις ως ($A \subseteq B$) && (A != B).

Εδώ όμως θα πρέπει να παρατηρήσουμε μερικά πράγματα:

- Οι τύποι των παραμέτρων βάζουν πολύ λίγους περιορισμούς: είναι απλώς *προσεγγιστές εισόδου*, δηλαδή ό,τι πιο γενικό. Πουθενά δεν φαίνεται ότι θα πρέπει να είναι προσεγγιστές συνόλου. Στην πραγματικότητα γίνεται σύγκριση των περιεχομένων δύο περιοχών που μπορεί να βρίσκονται σε οποιοδήποτε περιέχον STL, σε πίνακα ή ακόμη και στα δεδομένα εισόδου! Πράγματι, αν γράψουμε:

```
// . . .
   istream_iterator< int > cinIt( cin ), cinEoStream;
   if ( includes(intSet2.begin(), intSet2.end(),
                 cinIt, cinEoStream) )
      cout << "intSet2 includes input" << endl;
// . . .
```

και κατά την εκτέλεση πληκτρολογίσουμε:

```
-11 8 31 e
```

θα πάρουμε:

```
intSet2 includes input
```

Αν γράψουμε:

```
// . . .
   copy( intArr+4, intArr+7, ostream_iterator<int>(cout, " ") );
   cout << endl;
   if ( includes(intSet2.begin(), intSet2.end(),
                 intArr+4, intArr+7) )
      cout << "intSet2 includes [intArr+4, intArr+7)" << endl;
```

```
// . . .
```

θα πάρουμε:

```
-11 31 41
intSet2 includes [intArr+4, intArr+7)
```

Τέλος, αν γράψουμε:

```
deque<int> intDeque( intArr+1, intArr+6 );
sort( intDeque.begin(), intDeque.end() );
copy( intDeque.begin(), intDeque.end(),
      ostream_iterator<int>(cout, " ") );
cout << endl;
if ( includes(intSet2.begin(), intSet2.end(),
              intDeque.begin(), intDeque.end()) )
   cout << "intSet2 includes intDeque" << endl;
```

παίρνουμε:

```
-23 -11 8 31 72
intSet2 includes intDeque
```

- Πρόσεξε την τελευταία περίπτωση: πριν καλέσουμε την *includes()* καλούμε τη *sort()* για να ταξινομήσει τα περιεχόμενα της *intDeque*. Αυτό είναι ουσιώδες: Αν δεν το κάνουμε θα πάρουμε λάθος αποτελέσματα:

```
72 -23 8 -11 31
intSet2 does not include intDeque
```

Για να καταλάβεις τι γίνεται αντιγράφουμε –«ξεκαθαρίζοντας» κάπως– το περίγραμμα *includes()* από τη gcc (αρχείο `stl_algo.h`):[16]

```
 0: template< typename InputIter1, typename InputIter2 >
 1: bool includes( InputIter1 first1, InputIter1 last1,
 2:                InputIter2 first2, InputIter2 last2 )
 3: {
 4:    while( first1 != last1 && first2 != last2 )
 5:       if( *first2 < *first1 )
 6:          return false;
 7:       else if( *first1 < *first2 )
 8:          ++first1;
 9:       else
10:       {  ++first1; ++first2;  }
11:
12:    return first2 == last2;
13: }
```

Όπως καταλαβαίνεις, το πρόβλημα είναι η σύγκριση της γραμμής 5: "`34 < 72`" και δεν υπάρχει αν κάνουμε την ταξινόμηση.

Στη δεύτερη μορφή του περιγράμματος η γραμμή 5 γίνεται:

```
if( comp(*first2,*first1) )
```

και η γραμμή 7:

```
else if( comp(*first1,*first2) )
```

Προϋποθέσεις για τον αλγόριθμο *includes()*: Οι δύο περιοχές

- Περιέχουν στοιχεία του ίδιου τύπου.[17]
- Τα στοιχεία είναι ταξινομημένα.
- Η ταξινόμηση έγινε με το ίδιο κατηγόρημα *comp()*.

Τα παραπάνω ισχύουν και για τις πράξεις μεταξύ συνόλων που βλέπουμε στη συνέχεια. Αν βάλεις στο πρόγραμμά σου "`#include <algorithm>`" μπορείς να χρησιμοποιείς και τις:

---

[16] Κάτι σου θυμίζει; Ξαναδές τον αλγόριθμο της συγχώνευσης (§9.5.3).

[17] Και τα στοιχεία είναι μοναδικά για την καθεμιά; Η αλήθεια είναι ότι αλγόριθμος δουλεύει και χωρίς τη μοναδικότητα, π.χ. και για πολυσύνολα.

- (*std*::)*set_union* (ένωση συνόλων),
- (*std*::)*set_intersection* (τομή συνόλων),
- (*std*::)*set_difference* (διαφορά συνόλων) και
- (*std*::)*set_symmetric_difference* (συμμετρική διαφορά συνόλων).[18]

Με αυτές όμως έχουμε και ένα άλλο πρόβλημα. Ας πάρουμε, για παράδειγμα, το:

```
template< class InputIter1, class InputIter2, class OutputIterator >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result );
template< class InputIter1, class InputIter2,
          class OutputIterator, class Compare >
OutputIterator set_union( InputIter1 first1, InputIter1 last1,
                          InputIter2 first2, InputIter2 last2,
                          OutputIterator result, Compare comp );
```

Το νέο στοιχείο εδώ είναι ότι έχουμε *αποτέλεσμα* που είναι (κατ' αρχήν) *σύνολο*. Γιατί «κατ' αρχήν»; Διότι ναι μεν είναι σύνολο αλλά μπορείς να το αποθηκεύσεις όπου σε βολεύει.

Με την παράμετρο *result* περνούμε την *αρχή* της περιοχής όπου θα αποθηκευτεί το απότέλεσμα. Η συνάρτηση επιστρέφει τιμή-προσεγγιστή προς το *τέλος* της περιοχής όπου έγινε η αποθήκευση. *Δεν πρέπει να υπάρχει επικάλυψη της περιοχής που θα πάρει το αποτέλεσμα με οποιαδήποτε από τις περιοχές που ενώνονται.*

Ας πούμε ότι έχουμε δύο σύνολα:

```
set< int > intSet1, intSet2;
```

και τους δίνουμε τιμές { 4, 5, 6 } και { 2, 4, 6, 8, 10 } αντιστοίχως. Μπορούμε να πάρουμε την ένωσή τους κατ' ευθείαν στο *cout*:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           ostream_iterator<int>(cout, " ") );
cout << endl;
```

και να δούμε στην οθόνη μας:

```
2 4 5 6 8 10
```

Μπορούμε να πάρουμε την ένωσή τους στο:

```
vector<int> intVec1(11);
```

με την εντολή:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           intVec1.begin() );
copy( intVec1.begin(), intVec1.end(),
      ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
2 4 5 6 8 10 0 0 0 0 0
```

Τι είναι αυτά τα μηδενικά; Για τον πίνακα έχουμε κρατήσει 11 θέσεις ενώ η ένωση έχει μόνον 6. Δηλαδή αν δηλώσουμε:

```
vector<int> intVec1;
```

δεν θα βγούνε; Δεν θα βγούνε αλλά θα πρέπει να χρησιμοποιήσουμε τον προσαρμογέα που μάθαμε στην §26.1.3:

```
back_insert_iterator< vector<int> > destV( intVec1 );
```

και να δώσουμε:

---

[18] Η **συμμετρική διαφορά** δύο συνόλων A, B ορίζεται ως $A \Delta B = (A \setminus B) \cup (B \setminus A)$· δηλαδή τα στοιχεία του A που δεν ανήκουν στο B και τα στοιχεία του B που δεν ανήκουν στο A. Άρα: $A \Delta B = (A \cup B) \setminus (A \cap B)$.

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           destV );
```

Και αν θέλουμε να αποθηκεύσουμε το αποτέλεσμα σε ένα

```
set<int> intSet3;
```

θα κάνουμε το ίδιο; Όχι, αλλά κάτι παρόμοιο: Το περίγραμμα *back_insert_iterator* μπορεί να δώσει στιγμιότυπα για κλάσεις που έχουν την *push_back()*. Ένα άλλο περίγραμμα-προσαρμογέας που μπορεί να δώσει στιγμιότυπα για κάθε κλάση που έχει την *insert()* είναι ο *inserter*. Για να τον χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <iterator>`".

Αν δώσουμε:

```
set_union( intSet1.begin(), intSet1.end(),
           intSet2.begin(), intSet2.end(),
           inserter(intSet3, intSet3.begin()) );
```

η ένωση θα φυλαχθεί στο *intSet3*.

Με τον ίδιο τρόπο δηλώνονται και οι άλλες συναρτήσεις (περιγράμματα) για σύνολα. Για παράδειγμα, για την τομή έχουμε:

```
template< class InputIter1, class InputIter2, class OutputIter >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result );
template< class InputIter1, class InputIter2,
          class OutputIter, class Compare >
OutputIterator set_intersection(InputIter1 first1, InputIter1 last1,
                                InputIter2 first2, InputIter2 last2,
                                OutputIter result, Compare comp );
```

Ο χειρισμός τους γίνεται με τον ίδιο τρόπο.

26.3.2 Το Περίγραμμα "*map*"

Το περίγραμμα "*set*" δεν είναι το μοναδικό εργαλείο της STL για παράσταση συνόλου. Υπάρχει και η **απεικόνιση** (map) που έχει την επικεφαλίδα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class map;
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου

```
#include <map>
```

Σε σύγκριση με το "*set*" υπάρχει μια επί πλέον παράμετρος: "`class T`". Τι είναι αυτό; Τα περιεχόμενα του περιγράμματος είναι οργανωμένα σε ζεύγη τύπου:

```
pair< Key, T >
```

Ο πρώτος τύπος (*Key*) είναι ο γνωστός μας (από τα σύνολα) *τύπος κλειδιών* (key type)· ο δεύτερος (*T*) είναι ο **τύπος τιμών** (value type). Έτσι, οι δυο συνιστώσες του ζεύγους είναι:

```
Key first;
T   second;
```

Μήπως το "`map<Key, T>`" είναι ίδιο με το "`set< pair<Key, T> >`"; Όχι! Όταν το ζεύγος μπαίνει στο περιέχον (`map`) η θέση του στη διάταξη καθορίζεται από την τιμή του *first* που μπαίνει στο δένδρο αναζήτησης και *δεν* (πρέπει να) *αλλάζει*.[19] Στο *second* βάζουμε όλα τα *άλλα στοιχεία που έχουμε τη δυνατοτητα να τροποποιούμε*. Στο "`set< pair<Key, T> >`" δεν έχουμε δυνατότητα για οποιαδήποτε τροποποίηση.

[19] Το «πρέπει να» για τους μεταγλωττιστές –σαν τον Borland C++ v.5.5– που αφήνουν την περιφρούρηση της δομής του συνόλου αποκλειστικώς στον προγραμματιστή.

Παράδειγμα

Γυρνάμε πίσω, στο Project 4, στην κλάση *CourseCollection* και αναγνωρίζουμε ότι έχουμε ένα σύνολο αντικειμένων *Course*. Θα πρέπει να αντικαταστήσουμε τη δήλωση "`Course* ccArr`" με μια δήλωση *map* αφού κάνουμε την κατάλληλη προεργασία.

- Ο τύπος κλειδιών είναι έτοιμος:

```
struct CourseKey
{
   enum { cCodeSz = 8 };
   char s[cCodeSz];
   explicit CourseKey( string aKey="" )
   {  strncpy( s, aKey.c_str(), cCodeSz-1 );
      s[cCodeSz-1] = '\0';  }
}; // CourseKey
```

- Όλα τα υπόλοιπα μπαίνουν στον τύπο τιμών:

```
class CourseVal
{
public:
   enum { cTitleSz = 80, cCategSz = 4 };
   explicit CourseVal( string aTitle="" );
// . . .
   void display( ostream& tout ) const;
private:
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
// . . .
   unsigned int cNoOfStudents;    // αριθ. φοιτητών
}; // Course
```

- Ο τύπος των ζευγών τιμών θα είναι:

```
   pair< CourseKey, CourseVal >
```

- Αφού για τον *CourseKey* δεν έχουμε επιφορτώσει τον "`<`" θα πρέπει να γράψουμε το κατάλληλο συναρτησοειδές:

```
struct CodeLT
{
   bool operator()( const CourseKey& ck1, const CourseKey& ck2 )
   {  return ( strcmp(ck1.s, ck2.s) < 0 );  }
}; // CodeLT
```

Έτσι, θα δηλώσουμε το σύνολό μας ως:

```
   map< CourseKey, CourseVal, CodeLT >  ccCourses;
```

Αν έχουμε:

```
   CourseKey aCk( "ΕΥ02010" );
   CourseVal aCv( "Αντικειμενοστρεφής Προγραμματισμός (Θ)" );
```

μπορούμε να εισαγάγουμε στο *ccCourses*:

```
   ccCourses.insert( make_pair(aCk, aCv) );
```

Αν

```
   map<CourseKey, CourseVal, CodeLT>::iterator it;
```

και ο *it* δείχνει το αντικείμενο που βάλαμε μπορούμε να κάνουμε τροποποίηση του *ccCateg*:

```
   (it->second).setCateg( "ΜΕΥ" );
```

Ο τελεστής "`[]`" –που δεν επιφορτώσαμε για την *CourseCollection* διότι μας φάνηκε «παράξενο» το "`allCourses["ΕΥ02010"]`"– επιφορτώνεται για το *map*. Δες το παρακάτω

Παράδειγμα

Στο παρακάτω πρόγραμμα έχουμε μια απεικόνιση από τα ονόματα των μηνών στα πλήθη ημερών τους.

```
  0: #include <iostream>
  1: #include <string>
```

```
 2: #include <map>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:    string monthN[] = { "Jan", "Feb", "Mar", "Apr",
 9:                        "May", "Jun", "Jul", "Aug",
10:                        "Sep", "Oct", "Nov", "Dec" };
11:    unsigned int noOfDays[] = { 31, 28, 31, 30, 31, 30,
12:                                31, 31, 30, 31, 30, 31 };
13:    map< string, unsigned int > months;
14:
15:    for ( int k(0); k < 6; ++k )
16:    {
17:       pair<string, unsigned int> amp;
18:       amp.first = monthN[k];
19:       amp.second = noOfDays[k];
20:       months.insert( amp );
21:    }
22:    for ( int k(6); k < 12; ++k )
23:       months[monthN[k]] = noOfDays[k];
24:
25:    for ( map<string, unsigned int>::const_iterator
26:                                        cit( months.begin() );
27:          cit != months.end(); ++cit )
28:       cout << cit->first << ", " << cit->second << endl;
29: }
```

Στις γραμμές 15-21 εισάγουμε τους πρώτους 6 μήνες δημιουργώντας τα κατάλληλα ζεύγη `pair<string, unsigned int>` και χρησιμοποιώντας τη μέθοδο *insert()*.

Στις γραμμές 22-23 εισάγουμε τους τελευταίους 6 μήνες αλλιώς:

```
   months["Jul"] = 31;
   months["Aug"] = 31;
// . . .
   months["Dec"] = 31;
```

Το "`months["Jul"] = 31`" είναι ισοδύναμο με "`months.insert(make_pair ("Jul",31))`".

Γράφουμε στο *cout* το περιεχόμενο στηριγμένοι στο ότι ο προσεγγιστής *cit* δείχνει ζεύγος που έχει ως *first* το όνομα του μήνα και ως *second* το πλήθος των ημερών. Το πρόγραμμα θα δώσει:

```
Apr, 30
Aug, 31
Dec, 31
Feb, 28
Jan, 31
Jul, 31
Jun, 30
Mar, 31
May, 31
Nov, 30
Oct, 31
Sep, 30
```

Οι μήνες βγαίνουν κατ' αλφαβητική σειρά. Αν αντικαταστήσουμε τις γραμμές 25-28 με τις:

```
   for ( int k(0); k < 12; ++k )
      cout << monthN[k] << ", " << months[monthN[k]] << endl;
```

όπου χρησιμοποιούμε τον "`[]`", θα πάρουμε:

```
Jan, 31
Feb, 28
Mar, 31
Apr, 30
```

```
May, 31
Jun, 30
Jul, 31
Aug, 31
Sep, 30
Oct, 31
Nov, 30
Dec, 31
```

Εκτός από την *insert()* που είδαμε στα δύο παραδείγματα υπάρχουμε και για το *map* οι άλλες δύο μορφές που είδαμε για το *set*.

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν για τα στιγμιότυπα του *map* όπως για τα στιγμιότυπα του *set*.

26.3.3 Τα Περιγράμματα "*multiset*" και "*multimap*"

Βάζοντας στο πρόγραμμά σου την "`#include <set>`" έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα "*multiset*" με επικεφαλίδα:

```
template < class Key, class Compare = less<Key>,
           class Allocator = allocator<Key> >
class multiset;
```

Κάθε στιγμιότυπο του περιγράμματος έχει αντικείμενα που είναι **πολυσύνολα** (multisets, bags). Σε αντιδιαστολή με το σύνολο, ένα πολυσύνολο μπορεί να έχει πολλαπλά αντίγραφα των μελών του. Έτσι, ενώ η σχέση "*ανήκει*" ορίζεται και για πολυσύνολα ($x \in m$)[20], πιο πολύ νόημα έχει η συνάρτηση **πολλαπλότητα** (multiplicity, multitude) που στο *multiset* υλοποιείται με την *count()*.

Όπως στα σύνολα έτσι και εδώ με την:

```
   multiset< int >  mulSet1;
```

δημιουργούμε ένα κενό πολυσύνολο (⟦ ⟧). Οι εντολές

```
   cout << "#mulSet1 == " << mulSet1.size() << endl;
   if ( mulSet1.empty() )  cout<< "mulSet1 == [[]]" << endl;
                     else  cout<< "mulSet1 != [[]]" << endl;
```

δίνουν:

```
#mulSet1 == 0
mulSet1 == [[]]
```

Θέλοντας να παραστήσουμε το ⟦ 1, 2, 4, 3, 2, 4, 4 ⟧ στο πρόγραμμά μας δίνουμε:

```
   int intArr[] = { 1, 2, 4, 3, 2, 4, 4 };

   multiset< int >  mulSet2( intArr, intArr+7 );
   copy( mulSet2.begin(), mulSet2.end(),
                              ostream_iterator<int>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
1 2 2 3 4 4 4
```

Για να δούμε τις πολλαπλότητες των "**2**" και "**7**" στο *mulSet2* δίνουμε:

```
   cout << "mulSet2 # 2 = " << mulSet2.count(2) << endl;
   cout << "mulSet2 # 7 = " << mulSet2.count(7) << endl;
```

και παίρνουμε:

```
mulSet2 # 2 = 2
mulSet2 # 7 = 0
```

Για το *multiset* υπάρχουν τρεις μέθοδοι *insert()*:

---

[20] Συμβολισμός της γλώσσας προδιαγραφών Z.

```
multiset<K>::iterator multiset<K>::insert( const K& x );
multiset<K>::iterator multiset<K>::insert( multiset<K>::iterator pos,
                                           const K& x );
template < class InputIterator >
void insert( InputIterator first, InputIterator last );
```

Με την πρώτη μορφή εισάγουμε στο πολυσύνολο την τιμή *x*. Αν η τιμή υπάρχει ήδη θα αυξηθεί η πολλαπλότητά της κατά 1. Επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο.

Και με τη δεύτερη μορφή εισάγουμε στο πολυσύνολο την τιμή *x* και μας επιστρέφει προσεγγιστή προς το νεοεισαχθέν στοιχείο. Η πρώτη παράμετρος (*pos*) μπορεί να επιταχύνει την εισαγωγή, όπως λέγαμε και στα σύνολα.

Με την τρίτη μορφή εισάγουμε όλα τα στοιχεία της περιοχής [*first*, *last*).

Τα περιγράμματα *includes()*, *set_union()*, *set_intersection()*, *set_difference()* και *set_symmetric_difference()* δουλεύουν και για το *multiset*.

Παρατηρήσεις:▶

1. Ως *ένωση* (*U*) και *τομή* (*I*) δύο πολυσυνόλων M_1 και M_2 ορίζονται τα πολυσύνολα που έχουν $U \# v = max(M_1 \# v, M_2 \# v)$ και $I \# v = min(M_1 \# v, M_2 \# v)$.

Στη βιβλιογραφία (π.χ. στη γλώσσα Z) θα βρεις να αναφέρεται ως «ένωση» και αυτό που άλλοι ονομάζουν «άθροισμα πολυσυνόλων» (bag sum) και έχει $S \# v = M_1 \# v + M_2 \# v$.

2. Ως *διαφορά* (*D*) των πολυσυνόλων $M_1 \setminus M_2$ ορίζεται το πολυσύνολο που έχει $D \# v = (M_1 \# v > M_2 \# v)\ ?\ M_1 \# v - M_2 \# v : 0$.◀

Με την "`#include <map>`" στο πρόγραμμά σου έχεις τη δυνατότητα να χρησιμοποιήσεις και το περίγραμμα:

```
template < class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
```

Όπως στο "*map*" έτσι και στο "*multimap*" τα περιεχόμενα είναι οργανωμένα σε ζεύγη τύπου:

```
pair< Key, T >
```

26.3.4 Διάταξη Στοιχείων

Τα στοιχεία ενός συνόλου (ή ενός πολυσυνόλου) δεν είναι κατ' ανάγκη διαταγμένα. Αλλά, όπως θα παρατήρησες σε όλα τα παραδείγματα που δώσαμε τα στοιχεία συνόλων και πολυσυνόλων βγαίνουν ταξινομημένα κατ' αύξουσα τάξη χωρίς να το ζητήσουμε. Γιατί; Διότι *«για να έχουμε γρήγορη αναζήτηση τα κλειδιά οργανώνονται –με βάση τη διάταξη "<" που είδαμε πιο πάνω– σε ένα δένδρο δυαδικής αναζήτησης.»* Έτσι, όταν δηλώνεις σύνολο με στοιχεία τύπου *K* θα πρέπει στον τύπο αυτόν να είναι ορισμένη η διάταξη "<".

Πάντως είναι δυνατόν να ανατρέψεις αυτήν τη διάταξη ορίζοντας δική σου όπως θα δεις στο παρακάτω παράδειγμα:

```
int      intArr[] = { 34, 72, -23, 8, -11, 31, 41 };

set< int >  intSet( intArr, intArr+7 );
copy( intSet.begin(), intSet.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;

set< int, greater<int> > intSetG( intArr, intArr+7 );
copy( intSetG.begin(), intSetG.end(),
                              ostream_iterator<int>(cout, " ") );
cout << endl;
```

Αποτέλεσμα:

```
-23 -11 8 31 34 41 72
72 41 34 31 8 -11 -23
```

Το δεύτερο σύνολο μας δίνει τα μέλη του κατά φθίνουσα τάξη. Αυτό είναι αποτέλεσμα του συναρτησοειδούς που έχουμε ως δεύτερο όρισμα (`greater <int>`) στο στιγμιότυπο του περιγράμματος.

Για παράδειγμα δες πώς ορίζεται (στο `functional`) το περίγραμμα

```
template < class T >
struct greater : public binary_function<T, T, bool>
{
   bool operator() (const T& x, const T& y) const { return x > y; }
};
```

Μέσα στο πρόγραμμά σου μπορείς να γράψεις: `greater(5,2)` ή `greater(0.5, 1.2)` και να σου δώσουν τιμές `true` και `false` αντιστοίχως. Φυσικά, για να γίνει η εξειδίκευση θα πρέπει να ορίζεται ο τελεστής "`>`" για τον τύπο *T*.

Αν δεν ορίζεται ο τελεστής διάταξης που σε ενδιαφέρει τι κάνεις; Έχεις δύο επιλογές:

- Να τον επιφορτώσεις (αν φυσικά επιφορτώνεται).
- Να γράψεις το δικό σου συναρτησοειδές.

Για τη δεύτερη περίπτωση δες το παρακάτω παράδειγμα:

```
struct cstrGT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) > 0;
   }
};

int main()
{
   set<const char*, cstrGT> css;
   css.insert( "μια" );  css.insert( "κλάση" );
   css.insert( "στην" );  css.insert( "οποία" );
   css.insert( "επιφορτώνουμε" );  css.insert( "τον" );
   css.insert( "τελεστή" );  css.insert( "κλήσης" );
   css.insert( "συνάρτησης" );
   copy( css.begin(), css.end(),
                        ostream_iterator<const char*>(cout, " ") );
   cout << endl;
```

Αποτέλεσμα:

```
τον τελεστή συνάρτησης στην οποία μια κλήσης κλάση επιφορτώνουμε
```

Αν όμως ορίσουμε:

```
struct cstrLT
{
   bool operator()( const char cs1[], const char cs2[] )
   {
      return strcmp(cs1, cs2) < 0;
   }
};
```

δηλώσουμε

```
   set<const char*, cstrLT> css;
```

και εισαγάγουμε τις ίδιες λέξεις παίρνουμε:

```
επιφορτώνουμε κλάση κλήσης μια οποία στην συνάρτησης τελεστή τον
```

Δηλώνοντας:

```
set< T > s;
```

είναι σαν να δηλώνεις

```
set< T, less<T> > s;
```

με την προϋπόθεση ότι στον *T* είναι ορισμένος ο "`<`".

Αυτά που λέμε εδώ για τα στιγμιότυπα του "*set*" εφαρμόζονται και στα άλλα συνειρμικά περιέχοντα και όχι μόνον· τα είδαμε και στα περιγράμματα *min(), max(), sort()* κλπ. Δηλαδή, εκείνο το *comp()* είναι σαν τον "`<`"; Πρόσεξε:

- Όταν γράφουμε "`a < b`" εννοούμε ότι "το *a* είναι μικρότερο του *b*". Εννοούμε όμως και κάτι άλλο: αν έχουμε αύξουσα διάταξη το *a προηγείται του b*.
- Η παράσταση "`comp(a, b)`" έχει τιμή `true` αν και μόνον αν το *a προηγείται του b*. Δηλαδή το *comp()* καθορίζει τη διάταξη· το «μικρότερο» το ξεχνούμε.

Από την επιφόρτωση του "`<`" ή τον ορισμό του *comp()* έχουμε και υποκατάστατο του "`==`": "`!(a<b) && !(b<a)`" ή "`!comp(a,b) && !comp(b,a)`". Στην περίπτωση αυτή λέμε ότι τα κλειδιά *a* και *b* είναι **ισοδύναμα** (equivalent) κρατώντας τον όρο «*ίσα*» μόνο για την περίπτωση σύγκρισης με τον "`==`" (αν είναι ορισμένος).

Στις παρατηρήσεις για τη *lower_bound()* στην §26.1.3 γράφαμε «*Η comp πρέπει να είναι ίδια (γενικότερα: συμβατή) με αυτήν που χρησιμοποιήθηκε για την ταξινόμηση. Αλλιώς η αναζήτηση δεν γίνεται σωστά.*» Γενικώς:

- ***Αν μια συνάρτηση χρησιμοποιεί την ταξινόμηση των στοιχείων ενός περιέχοντος το συναρτησοειδές (comp ή less<T>) διάταξης θα πρέπει να είναι το ίδιο με αυτό που έγινε η ταξινόμηση.***

26.4 Ποιο Περιέχον να Διαλέξω;

Μετά την ερώτηση «*Ποια Ακολουθία να Διαλέξω;*» έρχεται η γενικότερη ερώτηση: «*Ποιο Περιέχον να Διαλέξω;*»

- Για πολλούς η απάντηση είναι η ίδια με αυτήν που θα έδιναν και στην πρώτη ερώτηση: «*vector και πάλι vector! Δεν νομίζω ότι υπάρχει κάτι άλλο!*»
- Άλλοι θα πουν: όπως λέγαμε στην §9.5.1 «*Πολύ συχνά, ένας πίνακας χρησιμοποιείται για την παράσταση κάποιου συνόλου.*» Θα πρέπει λοιπόν να χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα των περιεχομένων είναι κλειδιά ή
  - το "*map*".

Η δεύτερη άποψη φαίνεται πιο σωστή αφού έχουμε πιο εύκολη εισαγωγή και διαγραφή στοιχείων· θυμίσου ότι στο σημείο αυτό το "*vector*" υστερεί και ως προς το "*list*". Ακόμη, οι αναζητήσεις στα συνειρμικά περιέχοντα έχουν λογαριθμική πολυπλοκότητα.

Βέβαια και στο "*vector*" μπορείς να έχεις γρήγορες αναζητήσεις αν έχεις ταξινομήσει τα περιεχόμενά του. Ακόμη, το "*vector*" έχει το πλεονέκτημα της αποθήκευσης των στοιχείων του σε συναπτές θέσεις της μνήμης πράγμα που επιτρέπει –εκτός από τη δυαδική αναζήτηση– τη χρήση συναρτήσεων που έχουν γραφεί για απλούς πίνακες. Το μειονέκτημά του είναι η χρονοβόρα εισαγωγή/διαγραφή στοιχείων έτσι ώστε να μην καταστρέφεται η ταξινόμηση.

Ας προσπαθήσουμε να βγάλουμε μερικούς κανόνες για τις επιλογές μας.

Ένας κάπως επιφανειακός αλλά όχι λανθασμένος κανόνας είναι αυτός που είπαμε πιο πάνω:

- Αν έχουμε μια συλλογή αντικειμένων που έχει χαρακτηριστικά συνόλου (μοναδικότητα στοιχείων) χρησιμοποιούμε
  - το "*set*" αν ολόκληρα τα αντικείμενα είναι κλειδιά ή
  - το "*map*".
- Αν η συλλογή μας είναι πολυσύνολο τότε
  - αν έχουμε συχνές εισαγωγές/διαγραφές χρησιμοποιούμε "*multiset*" ή "*multimap*",
  - αλλιώς χρησιμοποιούμε (ταξινομημένο) "*vector*".

Αν σε ενδιαφέρει πολύ η ταχύτητα θα πρέπει να εξετάσεις την περίπτωση του ταξινομημένου "*vector*" και σε σχέση με τα "*set*"/"*map*". Φυσικά, σε μια τέτοια περίπτωση έχεις να πληρώσεις και κάτι ακόμη: είναι δική σου υποχρέωση να διασφαλίζεις τη μοναδικότητα των μελών του συνόλου.

26.5 Άλλα Περιγράμματα

Στη συνέχεια θα δούμε εν συντομία δύο περιγράμματα κλάσεων που δεν είναι περιέχοντα: *bitset*, *complex*.[21]

26.5.1 Το Περίγραμμα "*bitset*"

Βάζοντας στο πρόγραμμά σου την οδηγία "`#include <bitset>`" μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template < size_t N >
class bitset;
```

που είναι σαν τη δική μας "*Bitmap*" αλλά χωρίς περιορισμούς από το μέγεθος (σε δυαδικά ψηφία) του ακέραιου: το μέγεθος καθορίζεται από την παράμετρο *N* και μπορεί να είναι πολύ μεγάλο.

Έχει κάποια σχέση με το "`vector<bool>`"; Μοιάζει πολύ αλλά και διαφέρει· η βασική διαφορά είναι ότι το "*bitset*"

- δεν είναι δυναμικό –η τιμή της *N* καθορίζεται όταν γίνεται η μεταγλώττιση και μετά δεν αλλάζει–
- δεν έχει προσεγγιστές και φυσικά δεν έχει "*push_back()*".

Με μεθόδους επιφορτώνονται οι τελεστές "`&=`", "`|=`", "`^=`", "`~`", "`[]`", "`<<=`", "`>>=`","`==`", "`!=`" ενώ με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται οι "`&`", "`|`", "`^`".

Οι "`<<`", "`>>`" επιφορτώνονται δύο φορές

- με μεθόδους ως τελεστές ολίσθησης:
  ```
  bitset<N> operator<<(size_t pos) const;
  bitset<N> operator>>(size_t pos) const;
  ```
- με περιγράμματα καθολικών συναρτήσεων ως τελεστές εισόδου/εξόδου:
  ```
  template < size_t N >
  istream& operator>>( istream& is, bitset<N>& x );
  template < size_t N >
  ostream& operator<<( ostream& os, const bitset<N>& x );
  ```

Το περίγραμμα έχει αρκετούς δημιουργούς:

- Ερήμην δημιουργό:
  ```
  bitset();
  ```
  που ξεκινάει όλα τα ψηφία στο "`0`":
  ```
  bitset<12> bsd;
  cout << bsd << endl;
  ```
  Αποτέλεσμα:
  ```
  000000000000
  ```
- Δημιουργό με αρχική τιμή `unsigned long int`:
  ```
  bitset( unsigned long val );
  ```
  Ας τον δούμε με ένα παράδειγμα: οι
  ```
  bitset<12> bsu0( 21UL ), bsu1( 1000000UL );
  cout << bsu0 << endl << bsu1 << endl;
  ```

[21] Δεν θα δούμε το περίγραμμα *valarray* και τα σχετικά με αυτό.

```
bitset<32> bsu2( 1000000UL );
cout << bsu2 << endl;
```

δίνουν:

```
000000010101
001001000000
00000000000011110100001001000000
```

Στην πρώτη γραμμή βλέπουμε την παράσταση του "**21**" στο δυαδικό σύστημα (16+4+1) με 12 ψηφία (το λιγότερο σημαντικό ψηφίο στα δεξιά).

Στην τρίτη γραμμή βλέπεις το 1 εκατομμύριο σο δυαδικό σύστημα (32 ψηφία). Και στη δεύτερη τι έχουμε; Τα 12 λιγότερο σημαντικά ψηφία αυτού που βλέπουμε στην τρίτη γραμμή· τόσα χωράνε στη *bsu1*!

Καταλαβαίνεις λοιπόν πώς δουλεύει αυτός ο δημιουργός:

- Αν η δυαδική παράσταση του ορίσματος χωράει στα δυαδικά ψηφία που έχουμε (*N*) όλα πάνε καλά (με πιθανή συμπλήρωση με μηδενικά από αριστερά).
- Αν δεν χωράει αποθηκεύονται τα λιγότερο σημαντικά ψηφία που χωράνε.

- Δημιουργό με αρχική τιμή **string**:

  ```
  explicit bitset( const string& str, size_type pos=0, size_type n=npos );
  ```

  Το πρέπει να έχει μόνον τους χαρακτήρες "**0**" και "**1**". Αν βρεθεί κάτι άλλο ρίχνεται εξαίρεση (*std*::)*invalid_argument*. Μπορείς να ορίσεις και υποορμαθό (*n* χαρακτήρες ξεκινώντας από *pos*). Οι εντολές

```
bitset<12> bss0( string("100100101010") ),
           bss1( string("100100101") ),
           bss2( string("111111100100101010100101010") );
cout << bss0 << endl << bss1 << endl << bss2 << endl;
```

δίνουν:

```
100100101010
000100100101
010100101010
```

Στο *bss0* βάζουμε ως αρχική τιμή ορμαθό (με "**0**" και "**1**") με μήκος 12. Τα ψηφία γίνονται τιμές των δυαδικών ψηφίων της *bss0*.

Στη δεύτερη περίπτωση έχουμε ορμαθό με 9 ψηφία και συμπληρώνεται με τρία "**0**" στην αρχή.

Στην τελευταία περίπτωση ο ορμαθός έχει μήκος μεγαλύτερο από 12. Η παίρνει την τιμή της από τους *τελευταίους* 12 χαρακτήρες.

Ας δούμε τώρα τις αντιστοιχίες μεταξύ του (δικού μας) *Bitmap* και του *bitset*:

- Αντί για τη *bitValue()* έχουμε τον τελεστή "**[]**" που επιφορτώνεται με μεθόδους:

  ```
  bitset<N>::reference operator[]( size_t pos );
  bool operator[]( size_t pos ) const;
  ```

  Για τον τύπο "**bitset<N>::reference**" ισχύουν αυτά που είπαμε στην §26.2.1.2 για τον "**vector<bool>::reference**".

  Μπορείς ακόμη να πάρεις την τιμή του ψηφίου σε τύπο **bool** με τη μέθοδο

  ```
  bool test( size_t pos ) const;
  ```

- Αντί για τη *setBit()* έχουμε τη

  ```
  bitset<N>& set( size_t pos, int val=1 );
  ```

  Πάντως αυτή μπορεί να χρησιμοποιηθεί και για να μηδενίσουμε ένα ψηφίο αν κληθεί με δεύτερο όρισμα "**0**".

  Υπάρχει όμως και μια άλλη μορφή της *set()*

  ```
  bitset<N>& set();
  ```

  που βάζει τιμή "**1**" σε όλα τα ψηφία του ***this**.

- Αντί για την *clearBit()* υπάρχει η
  `bitset<N>& reset( size_t pos );`
  Είπαμε όμως ότι μπορείς να χρησιμοποιήσεις και τη *set()* με δεύτερο όρισμα "`0`".
  Και για τη *reset()* υπάρχει η μορφή
  `bitset<N>& reset();`
  που μηδενίζει όλα τα ψηφία του `*this`.
- Αντί για την *count1()* υπάρχει η
  `size_t count() const;`
- Αντί για τη *display()* χρησιμοποίησε τον τελεστή "`<<`" όπως κάναμε στα παραδείγματα που δώσαμε παραπάνω.
  Για τα τις *part()*, *firstBitSet()* δεν υπάρχουν αντίστοιχες.
  Άλλες μέθοδοι που υπάρχουν στο *bitset*:
- Μέθοδοι αντιστροφής δυαδικών ψηφίων: Η
  `bitset<N>& flip( size_t pos );`
  αντιστρέφει ("`0`"→"`1`" και "`1`"→"`0`") το ψηφίο στη θέση *pos* και επιστρέφει το `*this` και η
  `bitset<N>& flip();`
  επιστρέφει το `*this` αφού αντιστρέψει *όλα* τα ψηφία του. Οι "`bs1 = ~bs0`" και "`bs1 = bs0.flip()`" έχουν το ίδιο αποτέλεσμα.
- Μέθοδος μετατροπής σε `unsigned long int`:
  `unsigned long to_ulong() const;`
  Επιστρέφει την τιμή `unsigned long int` που αντιστοιχεί στα δυαδικά ψηφία που είναι αποθηκευμένα στο `*this`. Αν τα ψηφία είναι περισσότερα από αυτά που χρησιμοποιούνται για τον `long` ρίχνει εξαίρεση (*std*`::`)*overflow_error*.
- Μέθοδος μετατροπής σε `string`:
  `string to_string() const;`
  Επιστρέφει την τιμή `string` που προκύπτει από την αντιστοίχιση κάθε ψηφίου μηδέν στον χαρακτήρα `'0'` και κάθε ψηφίου ένα στον χαρακτήρα `'1'`.
- Μέθοδος:
  `bool any() const;`
  που επιστρέφει `true` αν έστω και ένα ψηφίο του `*this` έχει τιμή ένα.
- Μέθοδος:
  `bool none() const;`
  που επιστρέφει `true` αν δεν υπάρχει ψηφίο του `*this` με τιμή ένα.

Παρατήρηση:▶

Θα πεις τώρα: «Καλα, τα "*bits*" τα είδαμε· το "*set*" που είναι;» Ας δούμε ένα (όχι πολύ καλό) παράδειγμα: Γυρνάμε στο Project 4 και θέλουμε σε κάθε *Course* να βάλουμε και το σύνολο των φοιτητών που έχουν εγγραφεί σε αυτό. Τα στοιχεία των φοιτητών φυλάγονται σε *scNOfStudents* θέσεις του πίνακα scArr. Ένας τρόπος να λύσουμε το πρόβλημά μας είναι ο εξής: Σε κάθε αντικείμενο κλάσης *Course* προσθέτουμε ένα ακομή μέλος:

```
Bitset<allStudents.getNOfStudents()> cEnrolled;
```

Αν ο φοιτητής με τα στοιχεία του στη `scArr[k]` είναι γραμμένος στο μάθημα *crs* τότε το `crs.cEnrolled[k]` έχει τιμή "`1`" αλλιώς "`0`". Σε μια τέτοια περίπτωση αχρηστεύεται και το *cNoOfStudents* αφού θα μπορούμε να έχουμε:

```
unsigned int Course::getNoOfStudents() const
{  return cEnrolled,count();  }
```

Γιατί δεν είναι πολύ καλό το παράδειγμα; Διότι ο πίνακας φοιτητών δεν είναι και τοσο σταθερός: φοιτητές εισάγονται και διαγράφονται και οι θέσεις των στοιχείων στον πίνακα μπορεί να μεταβάλλονται.◀

26.5.2 Το Περίγραμμα "*complex*"

Βάζοντας στο πρόγραμμά σου "`#include <complex>`" μπορείς να χρησιμοποιήσεις το περίγραμμα

```
template< class T >
class complex
{
public:
   typedef T value_type;
   complex( const T& re=T(), const T& im=T() );
   complex( const complex& );
   template<class X>  complex( const complex<X>& );
   T real() const;
   T imag() const;
   complex<T>& operator=( const T& );
   complex<T>& operator+=( const T& );
   complex<T>& operator-=( const T& );
   complex<T>& operator*=( const T& );
   complex<T>& operator/=( const T& );
   complex& operator=( const complex& );
   template<class X> complex<T>& operator=( const complex<X>& );
   template<class X> complex<T>& operator+=( const complex<X>& );
   template<class X> complex<T>& operator-=( const complex<X>& );
   template<class X> complex<T>& operator*=( const complex<X>& );
   template<class X> complex<T>& operator/=( const complex<X>& );
}; // complex
```

για μιγαδικούς αριθμούς.

Στο `complex` υπάρχουν και τρεις εξειδικεύσεις του περιγράμματος για `float`, `double` και `long double`.

Ακόμη, με περιγράμματα καθολικών συναρτήσεων επιφορτώνονται:

- Οι τελεστές "`+`", "`-`", "`*`", "`/`".
- Οι συναρτήσεις *abs()* και *arg()* με τα

```
template< class T > T abs( const complex<T>& x )
{  return sqrt( x.real()*x.real()+x.imag()*x.imag() );  }
template< class T > T arg( const complex<T>& x )
{  return atan2( imag(x), real(x) )  }
```

  που μας δίνουν τα συστατικά της πολικής μορφής.

- Η συνάρτηση (περίγραμμα)

```
template< class T >
complex<T> polar( const T& rho, const T& theta = 0 );
```

  που μας δίνει έναν μιγαδικό από τα πολικά συστατικά του. Θα πρέπει να έχουμε:

$$x == polar(\, abs(x), arg(x)\,)$$

  αλλά, αφού έχουμε να κάνουμε με τιμές κινητής υποδιαστολής, αντί για το "`==`" καλύτερα να σκέφεται "≈".

- Η συνάρτηση

```
template< class T > complex<T> conj( const complex<T>& x );
```

  που μας δίνει τον συζυγή του *x*.

- Περιγράμματα συναρτήσεων για *sqrt()*, *pow()*, *cos()*, *sin()*, *tan()*, *exp()*, *log()*, *log10()*, *cosh()*, *sinh()*.

26.6 Τι (Πρέπει να) Έμαθες στο Κεφάλαιο Αυτό

Όταν λέμε STL εννοούμε *περιέχοντα*, *προσεγγιστές*, *συναρτησιακά αντικείμενα* και *αλγόριθμους* (και όχι απλώς το *vector* όπως πιστεύουν μερικοί).

Τα *περιέχοντα* της STL είναι *vector*, *list*, *deque* (ακολουθίες) και *set*, *map*, *multiset* και *multimap* (συνειρμικά). Το ότι μέχρι τώρα παριστάναμε όλες τις συλλογές αντικειμένων με πίνακες (απλούς ή δυναμικούς) δεν σημαίνει ότι από εδώ και πέρα θα τις παριστάνουμε με αντικείμενα στιγμιοτύπων του *vector*. Το περιέχον που επιλέγουμε εξαρτάται από το πρόβλημα που έχουμε να λύσουμε.

Οι *προσεγγιστές* είναι πολύτιμα εργαλεία για την αξιοποίηση των περιεχόντων αλλά και των αλγορίθμων. Με βάση τις δυνατότητές τους κατηγοριοποιούνται σε *προσεγγιστές εισόδου*, *εξόδου*, *πρόσθιους*, *αμφίδρομους* και *τυχαίας πρόσβασης*.

Τα *συναρτησιακά αντικείμενα* είναι απαραίτητα για την παραμετροποίηση των περιγραμμάτων κλάσεων και συναρτήσεων. Δεν χρειάζεται να γράφουμε τα πάντα: η STL μας δίνει μια μικρή αλλά πολύ χρήσιμη συλλογή.

Για να χρησιμοποιήσουμε τα περιέχοντα της STL, εκτός από τις μεθόδους τους, μας δίνονται και αρκετοί *αλγόριθμοι* (περιγράμματα συναρτήσεων). Πολλοί από αυτούς δρουν σε συλλογές αντικειμένων. Σε τέτοιες περιπτώσεις στις παραμέτρους της συνάρτησης περιλαμβάνονται προσεγγιστές *first*, *last* και υπονοείται επεξεργασία των αντικειμένων **it* όταν ο *it* διατρέχει την περιοχή [*first*, *last*).

Στις βιβλιοθήκες της C++ υπάρχουν και άλλα περιγράμματα κλάσεων εκτός από τα περιέχοντα. Εν συντομία είδαμε δύο τέτοια παραδείγματα: *bitset* και *complex*.

Ασκήσεις

26-1 Αν

```
vector < T > v1;
```

τι αποτέλεσμα έχουν οι εντολές:

```
set< T > s1( v1.begin(), v1.end() );
vector< T > v2( s1.begin(), s1.end() );
```

Πώς αλλιώς (χωρίς να χρησιμοποιήσεις σύνολο) θα μπορούσες να βάλεις στο *v1* το τελικό περιεχόμενο του *v2*;

26-2 Δύο συναρτήσεις χρήσιμες για πολυσύνολα είναι οι εξής:

- *eqmult()*: η *eqmult(v, M_1, M_2)* μας δίνει τιμή `true` αν δύο πολυσύνολα M_1, M_2 έχουν ίσα πλήθη του ατόμου *v*.
- **κλιμάκωση** (scaling): η $n \otimes M_1$ (*n*: φυσικός) μας δίνει πολυσύνολο που περιέχει τα στοιχεία του M_1 και μόνον αυτά αλλά σε *n*-πλάσιο πλήθος από ότι τα έχει το M_1.

Να τις υλοποιήσεις με τα κατάλληλα περιγράμματα συναρτήσεων που θα είναι δυνατόν να χρησιμοποιηθούν για στιγμιότυπα των *multiset*, *multimap*.

project

7

Φοιτητές και Μαθήματα με STL

Περιεχόμενα:

Prj07.1 Το Πρόγραμμα με STL I: *vector* **1040**
- Prj07.1.1 Κλάση *Course* 1040
- Prj07.1.2 Κλάση *CourseCollection* 1040
  - Prj07.1.2.1 Σχετικώς με τη *getArr()* 1043
- Prj07.1.3 Κλάση *Student* 1044
- Prj07.1.4 Κλάση *StudentCollection* 1046
- Prj07.1.5 Κλάση *StudentInCourse* 1049
- Prj07.1.6 Κλάση *StudentInCourseCollection* 1049
- Prj07.1.7 Ο Πίνακας-Ευρετήριο 1051

Prj07.2 Το Πρόγραμμα με STL II **1052**
- Prj07.2.1 Επιλογές 1052
  - Prj07.2.1.1 ccArr της *CourseCollection* 1052
  - Prj07.2.1.2 sCourses της *Student* 1053
  - Prj07.2.1.3 scArr της *StudentCollection* 1053
  - Prj07.2.1.4 siccArr της *StudentInCourseCollection* 1053
  - Prj07.2.1.5 Πίνακας-Ευρετήριο 1054
- Prj07.2.2 Κλάση *Course* 1054
- Prj07.2.3 Κλάση *CourseCollection* 1054
- Prj07.2.4 Κλάση *Student* 1057
- Prj07.2.5 Κλάση *StudentCollection* 1058
- Prj07.2.6 Κλάση *StudentInCourseCollection* 1061
  - Prj07.2.6.1 Δυο Λόγια για τη *SICKeyLT* 1064
- Prj07.2.7 Πίνακας – Ευρετήριο 1065

Prj07.3 Τι Είδαμε στα Δύο Παραδείγματα **1066**

Εισαγωγικό Σημείωμα:

Στόχος αυτού του Project είναι να ξαναδούμε το γνωστό μας πρόβλημα χρησιμοποιώντας αυτά που μάθαμε για την STL. Θα αλλάξουμε τη λύση που είδαμε στο Project 4 με δύο τρόπους:

- Στην πρώτη περίπτωση θα αλλάξουμε όλους τους δυναμικούς πίνακες σε *vector*.
- Στη δεύτερη περίπτωση θα επιλέγουμε το περιέχον που μας φαίνεται πιο κατάλληλο.

Και στις δύο περιπτώσεις, θα κάνουμε τις μετατροπές υπακούοντας στον εξής περιορισμό: *Δεν θα πρέπει να αλλάζει ο τρόπος που καλούνται οι μέθοδοι που είναι "*`public`*" ώστε* να μην χρειάζεται να αλλάξουμε τα προγράμματα που χρησιμοποιούν τις κλάσεις μας. Αυτό σημαίνει ότι οι μέθοδοι θα πρέπει:

- Να έχουν τις ίδιες προδιαγραφές (προϋπόθεση-απαίτηση).
- Να έχουν την ίδια επικεφαλίδα.

Για τις μεθόδους που είναι "`private`" είμαστε πιο ελαστικοί.

Όπως θα δεις στη συνέχεια, θα παραβιάσουμε τον παραπάνω περιορισμό μόνον στην περίπτωση μιας συγκεκριμένης λειτουργίας: στην εξαγωγή ολόκληρου πίνακα (όπως, για παράδειγμα, ο ccArr της *CourseCollection*).

Prj07.1 Το Πρόγραμμα με STL I: *vector*

Θα αλλάξουμε τη λύση του Project 4 αντικαθιστώντας με το κατάλληλο στιγμιότυπο του "`vector`" κάθε πίνακα που έχουμε στη λύση. Πέρα από αυτό μπορεί να αλλάζουμε και μερικά από αυτά που έχουμε γράψει με κάτι «καλό» από την STL.

Prj07.1.1 Κλάση *Course*

Η κλάση *Course* δεν έχει πίνακες και έτσι δεν αλλάζει.

Prj07.1.2 Κλάση *CourseCollection*

Εδώ η βασική αλλαγή είναι στη δήλωση του ccArr:

```
   vector<Course> ccArr;
```

Τα *ccIncr*, *ccNOfCourses*, *ccReserved* μας είναι άχρηστα. Η *getNOfCourses()* γίνεται:

```
   size_t getNOfCourses() const {  return ccArr.size();  }
```

Με τη *getArr()* τι θα κάνουμε; Μια λύση είναι:

```
   const Course* getArr() const {  return &ccArr[0];  }
```

Θα το ξανασυζητήσουμε…

Πέρα από τα παραπάνω, η πρώτη απλούστευση της κλάσης φαίνεται στον δημιουργό:

```
   CourseCollection() { };
```

και στον καταστροφέα που δεν τον χρειάζόμαστε! Εμείς όμως, κατά τη συνήθειά μας θα γράψουμε:

```
   ~CourseCollection() { };
```

Θα ξεκινήσουμε τις μετατροπές από την (εσωτερική) συνάρτηση-μέλος *findNdx()* που χρησιμοποιείται παντού. Αντί για τη *linSearch()* (για να γίνουμε «πιο STL») θα χρησιμοποιήσουμε την (*std*::)*find()*:

```
vector<Course>::iterator CourseCollection::findNdx( string code )
{
   vector<Course>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = ccArr.end();
   else
      ndx = find( ccArr.begin(), ccArr.end(), Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Η *findNdx()* επέστρεφε τιμή τύπου `int` που ήταν δείκτης στοιχείου του ccArr (δηλαδή `ccArr[ndx]`). Τώρα επιστρέφει έναν προσεγγιστή προς στοιχείο του πίνακα· έτσι:

όπου βάζαμε "`ccArr[ndx]`" θα βάζουμε "`*ndx`" και
όπου βάζαμε "`ccArr[ndx].`*X*" θα βάζουμε "`ndx->`*X*"

Ακόμη, οι πράξεις "`++ccNOfCourses`" και "`--ccNOfCourses`" δεν έχουν νόημα και θα πρέπει να αφαιρεθούν.

Παρατήρηση:▶
Για σύνδεση με τα προηγούμενα κρατούμε το όνομα "*findNdx*" που όμως είναι συντομογραφία του «*find index*» (βρες δείκτη). Τώρα –που η μέθοδος δεν επιστρέφει δείκτη αλλά προσεγγιστή– θα έπρεπε να αλλάξουμε το όνομα σε κάτι σαν "*findIt*" για το «*find iterator*» (βρες προσεγγιστή).◀

Ξεκινούμε με τη:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) != ccArr.end() );  };
```

Συνεχίζουμε με τη *delete1Course()*:

```
void CourseCollection::delete1Course( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx != ccArr.end() )  // υπάρχει
   {
      vector<Course>::iterator k;
      k = find_if( ccArr.begin(), ccArr.end(),
                   bind2nd(EqPrereq(),code) );
      if ( k != ccArr.end() )
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::prereqRef,
                                      code.c_str() );
      int enrStdnt( ndx->getNoOfStudents() );
      if ( enrStdnt > 0 ) // υπάρχουν εγγραφές φοιτητών
         throw CourseCollectionXptn( "delete1Course",
                                     CourseCollectionXptn::enrollRef,
                                      code.c_str(), enrStdnt );
      erase1Course( ndx );
   }
} // CourseCollection::delete1Course
```

Εδώ, πέρα από τις αλλαγές που έχουν σχέση με την αλλαγή τύπου του *ndx*, έχουμε και κάτι άλλο: ενσωματώσαμε αυτό που είδαμε στο παράδειγμα της §26.1.4. Εδώ βέβαια το *k* είναι τύπου `vector<Course>::iterator` και αντί για `ccArr`, `ccArr+ccNOfCourses` έχουμε `ccArr.begin()`, `ccArr.end()` αντιστοίχως. Η *EqPrereq* είναι ίδια:

```
struct EqPrereq :  binary_function< Course, string, bool >
{
   bool operator()( const Course& x, const string& y ) const
   {  return ( strcmp(x.getPrereq(), y.c_str()) == 0 );  }
}; // EqPrereq
```

Πρόσεξε ότι πρέπει να αλλάξει η

```
void CourseCollection::erase1Course( vector<Course>::iterator ndx )
{
  *ndx = ccArr.back();
  ccArr.pop_back();
} // CourseCollection::erase1Course
```

Και προχωρούμε στην *add1Course()*:

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   vector<Course>::iterator ndx( findNdx(aCourse.getCode()) );
   if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( strcmp(aCourse.getPrereq(), "") != 0 )  // υπάρχει
      {                                            // προαπαιτούμενο
         vector<Course>::iterator ndx(
                                     findNdx(aCourse.getPrereq()) );
         if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί του
                                   // προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                     CourseCollectionXptn::prereqRef,
                                         aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      insert1Course( aCourse );
      ndx = findNdx( aCourse.getCode() );
      ndx->clearStudents();
```

```
   }
} // CourseCollection::add1Course
```

όπου:

```
void CourseCollection::insert1Course( const Course& aCourse )
{
   try {  ccArr.push_back( aCourse );  }
   catch ( bad_alloc& )
   {  throw CourseCollectionXptn( "insert1Course",
                              CourseCollectionXptn::allocFailed );  }
} // CourseCollection::insert1Course
```

Εδώ, όπως περιμέναμε, η απλούστευση είναι εντυπωσιακή. Βέβαια, η πιθανότητα να «ξεμείνουμε» από μνήμη υπάρχει και παίρνουμε τα μέτρα μας.

Η τελευταία μέθοδος «ενός στοιχείου» είναι η:

```
const Course& CourseCollection::get1Course( string code ) const
{
   vector<Course>::const_iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return *ndx;
} // CourseCollection::get1Course
```

Πρόσεξε το "`const_iterator ndx`": αυτό είναι απαραίτητο συμπλήρωμα των "`const`" της επικεφαλίδας.

Πριν δούμε τις μεθόδους φύλαξης-φόρτωσης ας δούμε τη

```
void CourseCollection::swap( CourseCollection& rhs )
{
   ccArr.swap( rhs.ccArr );
} // CourseCollection::swap
```

που όπως βλέπεις απλουστεύθηκε.

Η φύλαξη γίνεται με τη:

```
void CourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::fileNotOpen );
   size_t  ccNOfCourses( ccArr.size() );
   bout.write( reinterpret_cast<const char*>(&ccNOfCourses),
               sizeof(ccNOfCourses) );
   for ( vector<Course>::const_iterator it(ccArr.begin());
         it != ccArr.end(); ++it )
      it->save( bout );
   if ( bout.fail() )
      throw  CourseCollectionXptn( "save",
                                   CourseCollectionXptn::cannotWrite );
} // CourseCollection::save
```

και η φόρτωση με τη:

```
void CourseCollection::load( ifstream& bin )
{
   CourseCollection tmp;
   size_t n;  //ccNOfCourses

   bin.read( reinterpret_cast<char*>(&n), sizeof(size_type) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Course oneCourse;
         oneCourse.load( bin );
         tmp.insert1Course( oneCourse );
      }
```

```
      if ( bin.fail() )
         throw  CourseCollectionXptn( "load",
                                      CourseCollectionXptn::cannotRead );
      swap( tmp );
   }
} // CourseCollection::load
```

Η *load()* μένει όπως ήταν. Αυτό οφείλεται στο ότι δηλώσαμε την *tmp* τύπου *CourseCollection*. Αν επιλέξουμε "`vector<Course> tmp`" τότε αλλάζουμε:

τη "`tmp.insert1Course(oneCourse)`" σε "`tmp.push_back(oneCourse)`"
και τη "`swap(tmp)`" σε "`ccArr.swap(tmp)`"

και η φόρτωση γίνεται κάπως πιο γρήγορα.

Η *display()* δεν αλλάζει παρά μόνο στη συνθήκη της **for**. Θα πρέπει να γίνει "`k < ccArr.size()`".

Οι *add1Student()* και *delete1Student()* –ενδιάμεσες προς τις μεθόδους της *Course* με το ίδιο όνομα– αλλάζουν όπως είπαμε στην αρχή της παραγράφου:

```
void CourseCollection::add1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->add1Student();
} // CourseCollection::add1Student
```

και

```
void CourseCollection::delete1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() ) // δεν υπάρχει τέτοιο μάθημα
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->delete1Student();
} // CourseCollection::delete1Student
```

Prj07.1.2.1 Σχετικώς με τη *getArr()*

Υποσχεθήκαμε να ξανασυζητήσουμε για τη μέθοδο *CourseCollection::getArr()* και αυτό θα κάνουμε αλλά πριν προχωρήσεις καλό είναι να γυρίσεις στην §20.7.5 και να διαβάσεις το σημείο "**5. Μέθοδος** *getAllStops()*".

Εκεί είχαμε καταλήξει ότι το καλύτερο που είχαμε να κάνουμε για να «βγάλουμε» έναν πίνακα από το αντικείμενο ήταν να επιστρέψουμε δυναμικό πίνακα, αντίγραφο του εσωτερικού. Βέβαια επισημάναμε τον κίνδυνο «διαρροής» μνήμης και στηρίζαμε τις ελπίδες μας για την αποφυγή της στις καλές συνήθειες των πεπειραμένων προγραμματιστών.[1]

Τώρα θα δούμε μια πιο σίγουρη λύση, με παράδειγμα εφαρμογής στον ccArr:

```
void CourseCollection::getArr( vector<Course>& crsArr ) const
{
   try  {  crsArr = ccArr;  }
   catch( bad_alloc& )
   {  throw CourseCollectionXptn( "getArr",
                                  CourseCollectionXptn::allocFailed );  }
} // CourseCollection::getArr
```

Αυτή η λύση είναι στην πραγματικότητα μια μορφή RAII. Όταν δώσεις:

```
   CourseCollection allCourses;
// . . .
   vector<Course> courseTbl;
```

[1] Που, όμως, όταν ρίχνονται εξαιρέσεις, μπορεί να μην είναι αρκετές.

```
// . . .
   allCourses.getArr( courseTbl );
```

η δυναμική μνήμη που παίρνουμε κρύβεται μέσα στην *courseTbl*. Όταν τελειώσει η ζωή της ο καταστροφέας της θα ανακυκλώσεί αυτομάτως και τη μνήμη.

Φυσικά, θα υπενθυμίσουμε ότι και αυτή η «καλύτερη λύση» κοστίζει σε υπολογιστικό χρόνο αφού έχουμε αντιγραφή πίνακα.

Με αυτόν τον τρόπο θα βγάζουμε όλους τους πίνακες στη συνέχεια. Στο σημείο αυτό θα αλλάξουν οι διεπαφές όλων των κλάσεων.

Prj07.1.3 Κλάση *Student*

Η *Student* αλλάζει αφού θα αντικαταστήσουμε τα «εργαλεία υλοποίησης» του δυναμικού πίνακα μαθημάτων:

```
   enum { sIncr = 3 };
   size_t             sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
   size_t             sReserved;
```

με τη

```
   vector<Course::CourseKey> sCourses;
```

Ύστερα από αυτό θα έχουμε:

```
   unsigned int getNoOfCourses() const {  return sCourses.size();  }
   void clearCourses() {  sCourses.clear();  sWH = 0;  }
```

Φυσικά, οι πράξεις "`++sNoOfCourses`" και "`--sNoOfCourses`" και εδώ δεν έχουν νόημα και θα πρέπει να αφαιρεθούν.

Γράφουμε τη *getCourses()* όπως κάναμε στην *CourseCollection* με τη *getArr()*:

```
void Student::getCourses( vector<Course::CourseKey>& crsTbl ) const
{
   try {  crsTbl = sCourses;  }
   catch( bad_alloc& )
   {  throw StudentXptn( sIdNum, "getCourses",
                         StudentXptn::allocFailed );  }
} // Student::getCourses
```

Αρχίζουμε με τους δημιουργούς: Ο ερήμην δημιουργός γίνεται:

```
Student::Student( int aIdNum )
   :  sWH( 0 )
{
   if ( aIdNum < 0 )
      throw StudentXptn( 0, "Student", StudentXptn::negIdNum,
                         aIdNum );
   sIdNum = aIdNum;
   sSurname[0] = '\0';
   sFirstname[0] = '\0';
}; // Student()
```

και ο δημιουργός αντιγραφής:

```
Student::Student( const Student& rhs )
{
   sIdNum = rhs.sIdNum;
   strcpy( sSurname, rhs.sSurname );
   strcpy( sFirstname, rhs.sFirstname );
   sWH = rhs.sWH;
   try  {  sCourses = rhs.sCourses;  }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "Student",
                         StudentXptn::allocFailed );
   }
} // Student( const Student& rhs )
```

Τώρα πρόσεξε: *δημουργός αντιγραφής δεν χρειάζεται κατ' αρχήν* αφού το περίγραμμα *vector* έχει (σωστό) δημιουργό αντιγραφής. Και γιατί το βάλαμε; Για να πιάνουμε και να αλλάζουμε τη *bad_alloc*.

Παρατήρηση:▶

Και για τον ερήμην δημιουργό δεν χρειάζεται να κάνουμε το ίδιο; Κατ' αρχήν: ναι!◀

Θα συνεχίσουμε με τη *findNdx()*· την παίρνουμε από την *CourseCollection*::*findNdx()* αν βάλουμε:

- Τύπο προσεγγιστή

  "`vector<Course::CourseKey>::iterator`"

  (αντί για "`vector<Course>::iterator`").
- Όνομα πίνακα "`sCourses`" (αντί για "`ccArr`").
- Αντικείμενο αναζήτησης "`Course::CourseKey(code)`" (αντί για "`Course(code)`").

```
vector<Course::CourseKey>::iterator Student::findNdx( const string& code )
{
   vector<Course::CourseKey>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = sCourses.end();
   else
      ndx = find( sCourses.begin(), sCourses.end(),
                  Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Και μετά από αυτήν οι τρεις μέθοδοι «ενός στοιχείου»:

```
   bool find1Course( const string& code ) const
   {  return ( findNdx(code) != sCourses.end() );  };

void Student::add1Course( const Course& oneCourse )
{
   if ( findNdx(oneCourse.getCode()) == sCourses.end() )
   {
      insert1Course( Course::CourseKey(oneCourse.getCode()) );
      sWH += oneCourse.getWH();
   }
} // Student::add1Course

void Student::delete1Course( const Course& oneCourse )
{
   vector<Course::CourseKey>::iterator
                                 ndx( findNdx(oneCourse.getCode()) );
   if ( ndx != sCourses.end() )  // υπάρχει
   {
      erase1Course( ndx );
      sWH -= oneCourse.getWH();
   }
} // Student::delete1Course
```

όπου:

```
void Student::insert1Course( const Course::CourseKey& aCode )
{
   try {  sCourses.push_back( aCode );  }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "insert1Course",
                         StudentXptn::allocFailed );
   }
} // Student::insert1Course

void Student::erase1Course( vector<Course::CourseKey>::iterator ndx)
{
   *ndx = sCourses.back();
   sCourses.pop_back();
```

```
} // Student::erase1Course
```

Από τις υπόλοιπες μεθόδους αλλάζουν και οι:

```
void Student::swap( Student& rhs )
{
   std::swap( sIdNum, rhs.sIdNum );

   char svs[sNameSz];
   strcpy( svs, sSurname );  strcpy( sSurname, rhs.sSurname );
   strcpy( rhs.sSurname, svs );

   strcpy( svs, sFirstname );
   strcpy( sFirstname, rhs.sFirstname );
   strcpy( rhs.sFirstname, svs );

   std::swap( sWH, rhs.sWH );

   sCourses.swap( rhs.sCourses );
} // Student::swap

void Student::save( ostream& bout ) const
{
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::fileNotOpen );
   bout.write( reinterpret_cast<const char*>(&sIdNum), sizeof(sIdNum) );
   bout.write( sSurname, sizeof(sSurname) );
   bout.write( sFirstname, sizeof(sFirstname) );
   bout.write( reinterpret_cast<const char*>(&sWH), sizeof(sWH) );
   size_t noOfCourses( sCourses.size() );
   bout.write( reinterpret_cast<const char*>(&noOfCourses),
               sizeof(noOfCourses) );// αριθμός μαθημάτων που δήλωσε
   for ( int k(0); k < noOfCourses; ++k )
      bout.write( sCourses[k].s, Course::cCodeSz );
   if ( bout.fail() )
      throw StudentXptn( sIdNum, "save", StudentXptn::cannotWrite );
} // Student::save

void Student::load( istream& bin )
{
   Student tmp;
   bin.read( reinterpret_cast<char*>(&tmp.sIdNum), sizeof(sIdNum) );
   if ( !bin.eof() )
   {
      bin.read( tmp.sSurname, sizeof(sSurname) );
      bin.read( tmp.sFirstname, sizeof(sFirstname) );
      bin.read( reinterpret_cast<char*>(&tmp.sWH), sizeof(sWH) );
      size_t noOfCourses;              // αριθ. μαθημ. που δήλωσε
      bin.read( reinterpret_cast<char*>(&noOfCourses), sizeof(noOfCourses) );
      for ( int k(0); k < noOfCourses; ++k )
      {
         char aKey[Course::cCodeSz];
         bin.read( aKey, Course::cCodeSz );
         tmp.sCourses.push_back( Course::CourseKey(aKey) );
      }
      if ( bin.fail() )
         throw StudentXptn( sIdNum, "load",
                            StudentXptn::cannotRead );
      swap( tmp );
   }
} // Student::load
```

Prj07.1.4 Κλάση *StudentCollection*

Εδώ, όπως κάναμε στις προηγούμενες κλάσεις, θα αντικαταστήσουμε τις

```
   enum { scIncr = 30 };
```

```
   Student*                scArr;
   size_t                  scNOfStudents;
   size_t                  scReserved;
```

με τη

```
   vector<Student>            scArr;
```

Και όπως μάθαμε, θα πρέπει να αλλάξουμε τις:

```
   size_t getNOfStudents() const {  return scArr.size();  }

void StudentCollection::getArr( vector<Student>& stdntTbl ) const
{
   try {  stdntTbl = scArr;  }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                              StudentCollectionXptn::allocFailed );  }
} // StudentCollection::getArr
```

Ο δημιουργός και ο καταστροφέας απλουστεύονται:

```
StudentCollection::StudentCollection()
{  scPAllEnrollments = 0;  }
   ~StudentCollection() { };
```

Η συνέχεια είναι γνωστή:

```
vector<Student>::iterator StudentCollection::findNdx( int aIdNum )
{
   vector<Student>::iterator ndx;
   if ( aIdNum <= 0 )
      ndx = scArr.end();
   else
      ndx = find( scArr.begin(), scArr.end(), Student(aIdNum) );
   return ndx;
} // StudentCollection::findNdx

   bool find1Student( int aIdNum ) const
   {  return ( findNdx(aIdNum) != scArr.end() );  };

const Student& StudentCollection::get1Student( int aIdNum ) const
{
   vector<Student>::const_iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "get1Student",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   return *ndx;
} // StudentCollection::get1Student

void StudentCollection::add1Student( const Student& aStudent )
{
   vector<Student>::iterator ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx == scArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
      insert1Student( aStudent );
      vector<Course::CourseKey> aStCourses;
      aStudent.getCourses( aStCourses );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                     StudentInCourse(aStudent.getIdNum(), aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Στην *add1Student()* πρόσεξε την αλλαγή στον χειρισμό του *aStCourses* λόγω της αλλαγής της *Student::getCourses()*.

```
void StudentCollection::delete1Student( int aIdNum )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx != scArr.end() )  // υπάρχει
   {
      if ( ndx->getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      erase1Student( ndx );
   }
} // StudentCollection::delete1Student

void StudentCollection::insert1Student( const Student& aStudent )
{
   try {  scArr.push_back( aStudent );  }
   catch( MyTmpltLibXptn& )
   {
      throw StudentCollectionXptn( "insert1Student",
                                   StudentCollectionXptn::allocFailed );
   }
} // StudentCollection::insert1Student

void StudentCollection::erase1Student(vector<Student>::iterator ndx)
{
   *ndx = scArr.back();
   scArr.pop_back();
} // StudentCollection::erase1Student
```

Εδώ έχουμε επιπλέον:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   ndx->add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   vector<Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound,
                                   aIdNum );
   ndx->delete1Course( aCourse );
} // StudentCollection::delete1Course
```

Τέλος, έχουμε τις:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   scArr.swap( rhs.scArr);
} // StudentCollection::swap

void StudentCollection::save( ofstream& bout, SIndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   size_t nOfStudents( scArr.size() );
   bout.write( reinterpret_cast<const char*>(&nOfStudents),
               sizeof(nOfStudents) );
```

```
   for ( int k(0); k < nOfStudents; ++k )
   {
      index[k].sIdNum = scArr[k].getIdNum();
      index[k].loc = bout.tellp();
      scArr[k].save( bout );
   }
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save

void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.insert1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Ένα ερώτημα για τη δεύτερη παράμετρο της *save()*: Δεν θα αλλάξουμε τον πίνακα του ευρετηρίου; Θα τον αλλάξουμε· είπαμε όμως να μην αλλάξουμε τις διεπαφές των κλάσεων. Θα τα πούμε στη συνέχεια…

Prj07.1.5 Κλάση *StudentInCourse*

Η κλάση *StudentInCourse* δεν χρειάζεται τροποποίηση.

Prj07.1.6 Κλάση *StudentInCourseCollection*

Εδώ, όπως καταλαβαίνεις, θα δηλώσουμε τον δυναμικό πίνακα ως:

```
   vector<StudentInCourse> siccArr;
```

Αυτό έχει τις εξής άμεσες επιπτώσεις:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection

   ~StudentInCourseCollection()  {  };

   size_t getNOfStudentInCourses() const  {  return siccArr.size();  }

void StudentInCourseCollection::getArr( vector<StudentInCourse>& sicTbl) const
{
   try {  sicTbl = siccArr;  }
   catch( bad_alloc& )
   {  throw StudentInCourseCollectionXptn( "getArr",
                     StudentInCourseCollectionXptn::allocFailed );  }
} // StudentInCourseCollection::getArr
```

Και –κατά τη συνήθειά μας– συνεχίζουμε με τη:

```
vector<StudentInCourse>::iterator
                StudentInCourseCollection::findNdx( int aIdNum, string code )
{
   vector<StudentInCourse>::iterator ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = siccArr.end();
   else
      ndx = find( siccArr.begin(), siccArr.end(),
                  StudentInCourse(aIdNum,code) );
   return ndx;
} // StudentInCourseCollection::findNdx
```

Μετά από αυτήν:

```
   bool find1StudentInCourse( int aIdNum, string code ) const
   {  return ( findNdx(aIdNum, code) != siccArr.end() );  }

const StudentInCourse&
          StudentInCourseCollection::get1StudentInCourse( int aIdNum,
                                                          string code ) const
{
   vector<StudentInCourse>::const_iterator
                                      ndx( findNdx(aIdNum, code) );
   if ( ndx == siccArr.end() ) // δεν υπάρχει στοιχείο
      throw StudentInCourseCollectionXptn( "get1StudentInCourse",
                           StudentInCourseCollectionXptn::notFound,
                           StudentInCourse::SICKey(aIdNum, code) );
   return *ndx;
} // StudentInCourseCollection::get1StudentInCourse
```

Στην *add1StudentInCourse()* το μόνο που αλλάζει είναι η συνθήκη της πρώτης **if**:

```
void StudentInCourseCollection::add1StudentInCourse(
                                         const StudentInCourse& aStdnInCrs )
{
   if ( findNdx( aStdnInCrs.getSIdNum(),     // δεν υπάρχει φοιτητής
                aStdnInCrs.getCCode()) == siccArr.end() )
   {
   // . . .
      insert1StudentInCourse( aStdnInCrs );
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Η *insert1StudentInCourse()* απλουστεύεται:

```
void StudentInCourseCollection::insert1StudentInCourse(
                                         const StudentInCourse& aStdnInCrs )
{
   try {  siccArr.push_back( aStdnInCrs );  }
   catch( MyTmpltLibXptn& )
   {
      throw StudentInCourseCollectionXptn( "insert1StudentInCourse",
                                 StudentInCourseCollectionXptn::allocFailed );
   }
} // StudentInCourseCollection::insert1StudentInCourse
```

Για τις διαγραφές έχουμε:

```
void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                        string code )
{
   vector<StudentInCourse>::iterator ndx( findNdx(aIdNum, code) );
   if ( ndx != siccArr.end() ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                      StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "delete1StudentInCourse",
                                    StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course( code) );
```

```
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      erase1StudentInCourse( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse
```

όπου:

```
void StudentInCourseCollection::erase1StudentInCourse(
                                      vector<StudentInCourse>::iterator ndx )
{
   *ndx = siccArr.back();
   siccArr.pop_back();
} // StudentInCourseCollection::erase1StudentInCourse
```

Και η *swapArr()* απλουστεύεται:

```
void StudentInCourseCollection::swapArr( StudentInCourseCollection& rhs )
{
   siccArr.swap( rhs.siccArr );
} // StudentInCourseCollection::swapArr
```

ενώ οι *save()* και *load()* γίνονται:

```
void StudentInCourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                StudentInCourseCollectionXptn::fileNotOpen );
   size_t nOfStudentInCourse( siccArr.size() );
   bout.write( reinterpret_cast<const char*>(&nOfStudentInCourse),
               sizeof(nOfStudentInCourse) );
   for ( int k(0); k < nOfStudentInCourse; ++k )
      siccArr[k].save( bout );
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                       StudentInCourseCollectionXptn::cannotWrite );
} // StudentInCourseCollection::save

void StudentInCourseCollection::load( ifstream& bin )
{
   StudentInCourseCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         StudentInCourse oneStudentInCourse;
         oneStudentInCourse.load( bin );
         tmp.insert1StudentInCourse( oneStudentInCourse );
      }
      if ( bin.fail() )
         throw  StudentInCourseCollectionXptn( "load",
                                 StudentInCourseCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentInCourseCollection::load
```

Prj07.1.7 Ο Πίνακας-Ευρετήριο

Είπαμε πιο πριν ότι δεν θα αλλάξουμε την επικεφαλίδα της *StudentCollection::save()* αλλά είπαμε στην αρχή ότι θα αλλάξουμε όλους τους πίνακες σε αντικείμενα στιγμιοτύπων του *vector*. Αυτά τα δύο είναι συμβατά; Ναι! Στη **main()** δίνουμε:

```
      vector<SIndexEntry> sIndex;
      try
      {  sIndex.reserve( allStudents.getNOfStudents() );  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
```

και στη συνέχεια:

```
    saveCollections( allCourses, allStudents, &sIndex[0], allEnrollments );
```

Η *saveCollections()* θα καλέσει:

```
allStudents.save( bout, sIndex );
```

Πρόσεξε το εξής: Προϋπόθεση της *StudentCollection::save()* είναι η ύπαρξη των στοιχείων του πίνακα sIndex. Αυτό μας το εξασφαλίζει η (επιτυχής) εκτέλεση της *sIndex.reserve()*. Αν δεν τη βάλουμε θα πρέπει να αλλάξουμε τη *save()* και να κάνουμε τις καταχωρίσεις με *push_back()*.

Prj07.2 Το Πρόγραμμα με STL II

Τώρα θα (ξανα)αλλάξουμε τη λύση του Project 4 αντικαθιστώντας κάθε πίνακα που έχουμε στη λύση με το καταλληλότερο –κατά περίπτωση– περιέχον.

Prj07.2.1 Επιλογές

Ξεκινούμε εξετάζοντας τους πίνακες έναν προς έναν για να δούμε με ποιο περιέχον θα αντικαταστήσουμε τον καθένα. Τι εξετάζουμε; Τη χρήση τους στα δύο προγράμματα του Project. Πάντως το δεύτερο πρόγραμμα το βλέπουμε γενικότερα, όπως θα έπρεπε να είναι, για τις ανάγκες της γραμματείας ενός τμήματος σπουδών.

Prj07.2.1.1 ccArr της *CourseCollection*

Σε αντικείμενο κλάσης *CourseCollection* έχουμε τον πίνακα μαθημάτων που υπάρχουν στο πρόγραμμα του τμήματος.

Ο πίνακας μαθημάτων είναι ένα *σύνολο* αντικειμένων τύπου *Course*. Ένα τέτοιο αντικείμενο είναι αρκετά μεγάλο (περί τα 100 B) ενώ το κλειδί είναι μικρό (8 B). Κατ' αρχήν θα πρέπει να παρασταθεί με στιγμιότυπο του *map*.

Ας δούμε όμως και τη χρήση του πίνακα:

- Στο 1ο Πρόγραμμα: ο πίνακας μαθημάτων, αφού φορτωθεί, χρησιμοποιείται μόνο για αναζητήσεις.
- Γενικότερα: Το πρόγραμμα ενός τμήματος αλλάζει μια φορά κάθε 3-4 χρόνια. Επομένως, η συνήθης χρήση του πίνακα μαθημάτων είναι η αναζήτηση πληροφοριών για τα μαθήματα που περιέχει.

Επειδή δεν έχουμε συχνή ενημέρωση του πίνακα θα εξετάσουμε και αυτό που λέμε στο τέλος της §26.4: ταξινομημένο *vector*. Δηλαδή:

- Στο 1ο Πρόγραμμα: ο πίνακας μαθημάτων, φορτώνεται σε ένα `vector <Course>`, ταξινομείται, στη συνέχεια οι αναζητήσεις γίνονται με τη *lower_bound()* και φυλάγεται ταξινομημένος.
- Στα άλλα προγράμματα ο πίνακας φορτώνεται ταξινομημένος και οι αναζητήσεις γίνονται με τη *lower_bound()*.

Στην περίπτωση αυτή θα έχουμε τα εξής προβλήματα:

- Όπως λέγαμε στην §26.4 «*είναι δική σου υποχρέωση να διασφαλίζεις τη μοναδικότητα των μελών του συνόλου.*» Αυτό το κάναμε ήδη στη λύση που έχουμε δώσει (§Prj07.1.2).
- Για να εκμεταλλευτούμε τα πλεονεκτήματα της επιλογής αυτής θα πρέπει να κάνουμε το εξής: Να κρατούμε τον πίνακα ταξινομημένο από την αρχή και να κάνουμε τις εισαγωγές με *insert()* στη σωστή θέση (και όχι με *push_back()*).

Παρατήρηση:▶
Γιατί να μην κάνουμε τις εισαγωγές στον πίνακα όπως την κάναμε στην πρώτη μετατροπή (με *push_back()*), στη συνέχεια να ταξινομήσουμε τον πίνακα και από εκεί και πέρα να ψάχνουμε με *lower_bound()*; Κάτι τέτοιο απαιτεί να έχουμε «διπλές» μεθόδους *findNdx()*, *find1Course()* και *get1Course()*. Αυτό είναι «εγγύηση» για δύσκολα λάθη στη συνέχεια! Ας το αφήσουμε καλύτερα…◀

Πιο πολύ για επίδειξη, θα επιλέξουμε τη λύση με το «ταξινομημένο *vector*». Αν θέλεις να δοκιμάσεις την απεικόνιση από τους κωδικούς μαθημάτων προς αντικείμενα *Course* καλύτερα να μιμηθείς αυτά που θα κάνουμε στη συνέχεια για τη *StudentCollection* και όχι αυτά που κάνουμε στο πρώτο παράδειγμα της §26.3.2.

Prj07.2.1.2 sCourses της *Student*

Εδώ έχουμε ένα πολύ μικρό *σύνολο* κωδικών μαθημάτων (κλειδιών).

Το σύνολο δημιουργείται στην αρχή του εξαμήνου και μπορεί να διορθωθεί μια φορά.[2] Χαρακτηριστική περίπτωση "*set*", αλλά μόνο για λόγους αρχής. Λόγω του μεγέθους του συνόλου δεν έχουμε κέρδος στις αναζητήσεις. Θα δεις ότι έχουμε κάτι μικρά κέρδη στις εισαγωγές και διαγραφές.

Prj07.2.1.3 scArr της *StudentCollection*

Αυτή η συλλογή έχει μερικές εκατοντάδες μέχρι μερικές χιλιάδες μέλη. Ενημερώσεις; Πολύ συχνές!

- Εισαγωγές με τις εγγραφές νέων φοιτητών.
- Διαγραφές με τις αποφοιτήσεις.
- Εισαγωγές με μετεγγραφές φοιτητών *από* άλλα τμήματα.
- Διαγραφές με μετεγγραφές φοιτητών *προς* άλλα τμήματα.
- Διαγραφές λόγω μη ανανέωσης εγγραφής.

Κάθε αντικείμενο της συλλογής είναι μοναδικό (τα στοιχεία ενός φοιτητή)· έχουμε δηλαδή ένα *σύνολο*.

Το μέγεθος του κλειδιού (αριθμός μητρώου του φοιτητή) είναι λιγότερο από 10% του μεγέθους του αντικειμενου. Εξ άλλου υπάρχουν και ενημερώσεις (π.χ. αλλαγές στο sCourses κάποιου φοιτητή) που δεν είναι εισαγωγές/διαγραφές.

Θα κάνουμε την υλοποίηση με "*map*", απεικόνιση των αριθμών μητρώου (κλειδιά) σε αντικείμενα *Student*.

Prj07.2.1.4 siccArr της *StudentInCourseCollection*

Εδώ έχουμε συλλογή με περίπου πενταπλάσιο αριθμό μελών από το προηγούμενο, αφού κάθε φοιτητής δηλώνει κατά μέσον όρο πέντε μαθήματα.

Κάθε αντικείμενο της συλλογής –δήλωση ενός φοιτητή για συμμετοχή σε ένα μάθημα για το τρέχον εξάμηνο– είναι μοναδικό· δηλαδή έχουμε και εδώ ένα *σύνολο*.

Ενημερώσεις; Αρκετές:

- Δηλώσεις φοιτητών για συμμετοχή στα μαθήματα στην αρχή του εξαμήνου.
- Διορθώσεις (διαγραφές-εγγραφές) των δηλώσεων.
- Ακυρώσεις των δηλώσεων όσων φοιτητών μετεγγράφονται *προς* άλλα τμήματα.

Στο τέλος του εξαμήνου έχουμε και άλλη ενημέρωση: εισαγωγή βαθμού.

Τι περιέχον θα επιλέξουμε; Πολλά τα βάλε-βγάλε άρα ούτε σκέψη για "*vector*".

---

[2] Δηλαδή νομίμως μια φορά. Διότι με τις «κλάψες» στη γραμματεία μπορεί να έχουμε και άλλες διορθώσεις.

Το κλειδί (αριθμός μητρώου, κωδικός μαθήματος) είναι –σε μέγεθος– το 75% του αντικειμένου. Μήπως το απλούστερο που έχουμε να επιλέξουμε είναι το "*set*"; Αν κάνουμε αυτήν την επιλογή, πώς θα βάζουμε τους βαθμούς; Κατ' ανάγκη με τις πράξεις: «βγάλε-διάγραψε-διόρθωσε-ξαναβάλε». Αν επιλέξουμε "*map*" θα έχουμε τη δυνατότητα για ενημέρωση επί τόπου.

Θα επιλέξουμε "*map*".

Prj07.2.1.5 Πίνακας–Ευρετήριο

Πόσο μας απλούστευσε το πρόγραμμα η αλλαγή που κάναμε στην §Prj07.1.7; Καθόλου! Ο απλός δυναμικός πίνακας είναι εξ ίσου απλός (αν όχι απλούστερος) με το στιγμιότυπο του *vector*. Φυσικά αυτό δεν είναι περίεργο:

- παίρνουμε όλη τη μνήμη που χρειαζόμαστε με μια `new[]` και την ανακυκλώνουμε με μια `delete[]`,
- συμπληρώνουμε τα στοιχεία με πολύ απλό τρόπο,
- έχουμε τη δυνατότητα να κάνουμε τη φύλαξη του με μια *write()*.

Κατ' αρχή λοιπόν δεν θα αλλάξουμε τη διαχείριση αυτού του πίνακα. Αλλά ας δούμε τι κάνουμε (ξανά και ξανά) στο δεύτερο πρόγραμμα:

- παίρνουμε έναν αριθμό μητρώου φοιτητή,
- βρίσκουμε από το ευρετήριο τη θέση του αντίστοιχου αντικειμένου στο αρχείο με τα στοιχεία των φοιτητών,
- το φορτώνουμε και
- το δείχνουμε.

Είπαμε όμως ότι η συλλογή στοιχείων φοιτητών «*έχει μερικές εκατοντάδες μέχρι μερικές χιλιάδες μέλη.*» Εδώ η ταχύτητα αναζήτησης έχει νόημα.[3] Τι να κάνουμε; Δεν θα αλλάξουμε περιέχον αλλά θα ταξινομήσουμε τον πίνακα (κατ' αύξοντα αριθμό μητρώου) ώστε να υπάρχει δυνατότητα δυαδικής αναζήτησης από τα προγράμματα εφαρμογής που τον χρησιμοποιούν.

Prj07.2.2 Κλάση *Course*

Όπως θα δεις παρακάτω, η *CourseCollection* θέτει απαίτηση για επιφόρτωση του τελεστή "<" για την *Course*: [4]

```
bool operator<( const Course::CourseKey& lhs, const Course::CourseKey& rhs )
{  return ( strcmp(lhs.s, rhs.s) < 0 );  }

bool operator<( const Course& lhs, const Course& rhs )
{  return ( Course::CourseKey(lhs.getCode()) <
            Course::CourseKey(rhs.getCode()) );  }
```

Prj07.2.3 Κλάση *CourseCollection*

Όπως στην §Prj07.1.2 έτσι και εδώ δηλώνουμε:

```
   vector<Course> ccArr;
```

[3] Θα πεις: και αυτό «κατ' αρχήν» διότι το βήμα «*παίρνουμε έναν αριθμό μητρώου φοιτητή*» εκτελείται με ανθρώπινες ταχύτητες· και θα έχεις δίκιο.

[4] Για λόγους συνέπειας με τους κανόνες μας, βάζουμε ακόμη:

```
bool operator>=( const Course::CourseKey& lhs, const Course::CourseKey& rhs)
{  return ( !(lhs < rhs) );  }
bool operator>=( const Course& lhs, const Course& rhs )
{  return ( !(lhs < rhs) );  }
```

και έχουμε:

```
   CourseCollection() { };
   ~CourseCollection() { };
   size_t getNOfCourses() const {  return ccArr.size();  }
   const Course* getArr() const {  return &ccArr[0];  }
```

Τη *getArr()* θα την αλλάξουμε όπως στην §Prj07.1.2.1.

```
void CourseCollection::getArr( vector<Course>& crsArr ) const
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
```

Να δούμε όμως τι γίνεται με τη *findNdx()*. Τώρα δεν θα καλέσουμε τη *find()* αλλά τη *lower_bound()*:

```
vector<Course>::iterator CourseCollection::findNdx( const string& code )
{
   vector<Course>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = ccArr.end();
   else
      ndx = lower_bound( ccArr.begin(), ccArr.end(), Course(code) );
   return ndx;
} // CourseCollection::findNdx
```

Πρόσεξε τώρα: η *lower_bound()* θα μας επιστρέψει προσεγγιστή προς το αντικείμενο που ψάχνουμε (αν υπάρχει) ή προς τη θέση που πρέπει να εισαχθεί (αν δεν υπάρχει). Επομένως η *find1Course()* θα γίνει έτσι:

```
bool CourseCollection::find1Course( const string& code ) const
{
   vector<Course>::iterator ndx( findNdx(code) );
   return ( (ndx != ccArr.end()) && (*ndx == Course(code)) );
} // CourseCollection::find1Course
```

Η συνθήκη στη `return` είναι η διατύπωση της απόφανσης «*το βρήκα*». Όπως λέγαμε στην §5.6, αν η "`ndx != ccArr.end()`" υπολογιστεί `false` η συνάρτηση επιστρέφει `false` χωρίς να κάνει την (απαγορευμένη) αποπαραπομπή του *ndx*. Η διατύπωση της «*δεν το βρήκα*» είναι η άρνηση της παράστασης της `return` που ισοδυναμεί με:

"`ndx == ccArr.end() || (*ndx != Course(code))`"

Και εδώ δεν κινδυνεύουμε από απαγορευμένη αποπαραπομπή.

Για να χρησιμοποιήσουμε αυτή τη μορφή της *lower_bound()* χρειάζεται να έχουμε ορίσει τον "`<`" για την *Course*.

Η *CourseCollection::delete1Course()* είναι ίδια με αυτήν της §Prj07.1.2 με μια μόνο διαφορά: στη συνθήκη της **if** που ελέγχει αν υπάρχει το προς διαγραφή αντικείμενο που τώρα γίνεται:

```
   if ( ndx != ccArr.end() && (*ndx == Course(code)) )  // υπάρχει
```

Ίδια είναι και η *EqPrereq*. Πιο σοβαρή είναι η διαφορά στην *erase1Course()* που τώρα γίνεται:

```
void CourseCollection::erase1Course( vector<Course>::iterator ndx )
{
   ccArr.erase( ndx );
} // CourseCollection::erase1Course
```

Γράφουμε ολόκληρη την *add1Course()* στη νέα της μορφή:

```
void CourseCollection::add1Course( const Course& aCourse )
{
   if ( strlen(aCourse.getCode()) != Course::cCodeSz-1 )
      throw CourseCollectionXptn( "add1Course",
                                  CourseCollectionXptn::entity );
   vector<Course>::iterator ndx( findNdx(aCourse.getCode()) );
   if ( ndx==ccArr.end() || (*ndx!=aCourse) )
   {                                      // δεν βρέθηκε το κλειδί
      if ( strcmp(aCourse.getPrereq(), "") != 0 )  // υπάρχει
      {                                            // προαπαιτούμενο
```

```
         vector<Course>::iterator ndx(findNdx(aCourse.getPrereq()));
         if ( ndx == ccArr.end() ) // δεν βρέθηκε το κλειδί του
                                   // προαπαιτούμενου
            throw CourseCollectionXptn( "add1Course",
                                        CourseCollectionXptn::prereqRef,
                                            aCourse.getPrereq() );
      }
   // δεν υπάρχει προαπαιτούμενο ή υπάρχει και βρέθηκε στον πίνακα
      ndx = insert1Course( ndx, aCourse );
      ndx->clearStudents();
   }
} // CourseCollection::add1Course
```

Ποιες είναι οι διαφορές;

- Η συνθήκη που ελέγχει την τιμή του *ndx*.
- Αλλάξαμε την *insert1Course()* ώστε να επιστρέφει προσεγγιστή προς το αντικείμενο που εισάχθηκε στον πίνακα όπως τον επιστρέφει η *ccArr.insert()*. Έτσι, δεν είναι απαραίτητο να αναζητήσουμε το αντικείμενο που μόλις βάλαμε στον πίνακα.

```
vector<Course>::iterator
               CourseCollection::insert1Course( vector<Course>::iterator ndx,
                                                const Course& aCourse )
{
   try {  return ccArr.insert( ndx, aCourse );  }
   catch ( bad_alloc& )
   {  throw CourseCollectionXptn( "insert1Course",
                             CourseCollectionXptn::allocFailed );  }
} // CourseCollection::insert1Course
```

Η *get1Course()* γίνεται:

```
const Course& CourseCollection::get1Course( string code ) const
{
   vector<Course>::const_iterator ndx( findNdx(code) );
   if ( ndx == ccArr.end() || (*ndx != Course(code)) )
      throw CourseCollectionXptn( "get1Course",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   return *ndx;
} // CourseCollection::get1Course
```

Δηλαδή, το μόνο που άλλαξε είναι ο έλεγχος του *ndx* στην **if**.

Το ίδιο ισχύει και για τις επόμενες:

```
void CourseCollection::add1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx==ccArr.end() || (*ndx!=Course(code)) )
      throw CourseCollectionXptn( "add1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->add1Student();
} // CourseCollection::add1Student

void CourseCollection::delete1Student( string code )
{
   vector<Course>::iterator ndx( findNdx(code) );
   if ( ndx==ccArr.end() || (*ndx!=Course(code)) )
      throw CourseCollectionXptn( "delete1Student",
                                  CourseCollectionXptn::notFound,
                                  code.c_str() );
   ndx->delete1Student();
} // CourseCollection::delete1Student
```

Η *swap()* γίνεται όπως στην §Prj07.1.2· το ίδιο και η *save()*.

Η *load()* πρέπει να αλλάξει λίγο: αφού αλλάξαμε την *insert1Course()* αντί "`tmp.insert1-Course(oneCourse)`" θα πρέπει να δώσουμε "`tmp.add1Course( oneCourse )`".

Prj07.2.4 Κλάση *Student*

Για το σύνολο κωδικών μαθημάτων δηλώνουμε:

```
   set<Course::CourseKey> sCourses;
```

και ύστερα από αυτό έχουμε (όπως στην §Prj07.1.3):

```
   unsigned int getNoOfCourses() const {  return sCourses.size();  }
   void clearCourses() {  sCourses.clear();  sWH = 0;  }
```

Η *getCourses()* αλλάζει λίγο:

```
void Student::getCourses( vector<Course::CourseKey>& crsTbl ) const
{
   try
   {
      crsTbl.clear();
      for ( set<Course::CourseKey>::const_iterator it(sCourses.begin());
            it != sCourses.end(); ++it )
         crsTbl.push_back( *it );
   }
   catch( bad_alloc& )
   {  throw StudentXptn( sIdNum, "getCourses", StudentXptn::allocFailed );  }
} // Student::getCourses
```

Τώρα δεν μπορούμε να περάσουμε τους κωδικούς με μια εκχώρηση· πρέπει να διασχίσουμε όλο το σύνολο και να τους περνούμε έναν προς έναν.

Οι δύο δημιουργοί είναι σαν αυτούς που γράψαμε στην §Prj07.1.3, τουλάχιστον οπτικώς. Αλλά

- Στον ερήμην δημιουργό δεν φαίνεται ότι δημιουργείται το (κενό σύνολο) *sCourses*.
- Στον δημιουργο αντιγραφής η "`sCourses = rhs.sCourses`" είναι εκχώρηση με σύνολα.

Να επισημάνουμε ότι ούτε εδώ χρειάζεται δημιουργός αντιγραφής αλλά τον αφήνουμε για να πιάνει και να αλλάζει την (πιθανή) *bad_alloc*. Για τον ίδιο λόγο (και μόνον) αφήνουμε και τον `Student::operator=()`. Όσο για τον καταστροφέα, αυτός δεν έχει τι να κάνει:

```
   ~Student() { };
```

Και προχωρούμε στη *findNdx()*. Μπορούμε να την πάρουμε από αυτήν της §Prj07.1.3 αλλάζοντας το "`vector<Course::CourseKey>::iterator`" σε "`set<Course::CourseKey>::iterator`"; Κατ' αρχήν ναι. Αλλά είναι προτιμότερο να κάνουμε άλλη μια αλλαγή:

```
set<Course::CourseKey>::iterator Student::findNdx( const string& code )
{
   set<Course::CourseKey>::iterator ndx;
   if ( code.length() != Course::cCodeSz-1 )
      ndx = sCourses.end();
   else
      ndx = sCourses.find( Course::CourseKey(code) );
   return ndx;
} // Student::findNdx
```

Η *sCourses.find()* ψάχνει σε χρόνο *O*(*log*(*sCourses.size()*)) αντί για το *O*(*sCourses.size()*) της αναζήτησης της *std::find()*.

Η *find1Course()* γίνεται, όπως και στην §Prj07.1.3:

```
   bool find1Course( const string& code )
   {  return ( findNdx(code) != sCourses.end() );  };
```

Θα αλλάξουμε όμως τις *add1Course()* και *delete1Course()* αχρηστεύοντας τις (εσωτερικές) *insert1Course()* και *erase1Course()*:

```
typedef pair< set<Course::CourseKey>::iterator, bool > InsRetType;

void Student::add1Course( const Course& oneCourse )
{
   try
   {
      InsRetType rv(sCourses.insert(Course::CourseKey(oneCourse.getCode())) );
```

```
      if ( rv.second )  sWH += oneCourse.getWH();
   }
   catch( bad_alloc& )
   {
      throw StudentXptn( sIdNum, "insert1Course", StudentXptn::allocFailed );
   }
} // Student::add1Course
```

Να δούμε τι κάναμε εδώ: Στην §26.3.1 είπαμε ότι η μέθοδος *insert()* του *set* (1η μορφή) παίρνει ως όρισμα το κλειδί που θέλουμε να εισαγάγουμε –στην περίπτωσή μας: `Course::CourseKey(oneCourse.getCode())`– και επιστρέφει μια τιμή *pair* (στην περίπτωσή μας: *rv*):

- Το πρώτο μέλος της (*rv.first*) είναι προσεγγιστής του τύπου συνόλου που δουλεύουμε (`set<Course::CourseKey>::iterator`) προς το κλειδί όπως βρίσκεται μέσα στο σύνολο.
- Το δεύτερο μέλος της (*rv.second*) είναι τιμή `bool` που έχει τιμή `true` αν έγινε η εισαγωγή ή`false` αν δεν έγινε εισαγωγή διότι η τιμή υπήρχε.

Αν το *rv.second* είναι `true` προσθέτουμε στις εβδομαδιαίες ώρες του φοιτητή τις εβδομαδιαίες ώρες του μαθήματος.

Παρομοίως, στη *delete1Course()*, αξιοποιούμε την τιμή που επιστρέφει η *erase()* του *set*:

```
void Student::delete1Course( const Course& oneCourse )
{
   size_t erased( sCourses.erase(Course::CourseKey(oneCourse.getCode())) );
   if ( erased > 0 )  sWH -= oneCourse.getWH();
} // Student::delete1Course
```

Φυλάγουμε την τιμή που επιστρέφει η *sCourses.erase()* στη μεταβλητή *erased*.

- Αν το κλειδί δεν υπήρχε στο σύνολο τότε η τιμή της γίνεται "`0`".
- Αλλιώς, αν υπήρχε, γίνεται "`1`" και επειδή ισχύει η "`erased > 0`" αφαιρούμε από τις εβδομαδιαίες ώρες του φοιτητή τις εβδομαδιαίες ώρες του μαθήματος.

Η *swap()* παραμένει όπως έγινε στην §Prj07.1.3 αλλά η εντολή "`sCourses.swap(rhs.sCourses)`" ανταλλάσσει τιμές συνόλων.

Στη *save()* υπάρχει μια υποχρεωτική αλλαγή: Η **for** δεν μπορεί να ελέγχεται πια με τον δείκτη του στοιχείου πίνακα· θα ελέγχεται από προσεγγιστή που διατρέχει ολόκληρο σύνολο:

```
   for( set<Course::CourseKey>::const_iterator it(sCourses.begin());
        it != sCourses.end(); ++it )
      bout.write( it->s, Course::cCodeSz );
```

Παρόμοια αλλαγή θα πρέπει να γίνει και στη *display()*.

Στη *load()*, βρίσκουμε τη διαφορά όταν φτάνουμε στην ανάγνωση των κωδικών μαθημάτων:

```
      for ( int k(0); k < noOfCourses; ++k )
      {
         char aKey[Course::cCodeSz];
         bin.read( aKey, Course::cCodeSz );
         tmp.sCourses.insert( Course::CourseKey(aKey) );
      }
```

Λόγω της αλλαγής περιέχοντος, η κλήση της *tmp.sCourses.insert()* αντικαθιστά την κλήση της *tmp.sCourses.push_back()*.

Prj07.2.5 Κλάση *StudentCollection*

Είπαμε στην §Prj07.1.1.4 ότι «*Θα κάνουμε την υλοποίηση με "map", απεικόνιση των αριθμών μητρώου (κλειδιά) σε αντικείμενα Student.*» Θα δηλώσουμε:

```
   map< unsigned int, Student > scArr;
```

Ο τύπος κλειδιών είναι "`unsigned int`" και ο τύπος τιμών "`Student`". Θα πεις: αφού το κλειδί είναι ο αριθμός μητρώου και αφού αυτό υπάρχει μέσα στο *Student* (μέλος *sIdNum*)

δεν έχουμε πλεονασμό (που, όπως είναι γνωστό, είναι επικίνδυνος); Έχουμε! Αλλά, θα είμαστε προσεκτικοί στους χειρισμούς μας. Θα κάνουμε εισαγωγές στοιχείων στο scArr με την:

"`scArr[aStudent.getIdNum()] = aStudent`"

ή με την:

"`scArr.insert( make_pair(aStudent.getIdNum(),aStudent) )`"

Σε σχέση με την §Prj07.1.4, δεν έχουμε αλλαγή στη

```
   size_t getNOfStudents() const {  return scArr.size();  }
```

αλλά έχουμε αλλαγή στη

```
void StudentCollection::getArr( vector<Student>& stdntTbl ) const
{
   try
   {
      for ( map<unsigned int,Student>::const_iterator it(scArr.begin());
            it != scArr.end();  ++it )
         stdntTbl.push_back( it->second );
   }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                                   StudentCollectionXptn::allocFailed );  }
} // StudentCollection::getArr
```

Ο *it* στη **for** δείχνει ένα ζεύγος (κλειδί, αντικείμενο). Το "`it->second`" είναι το αντικείμενο με τα στοιχεία φοιτητή. Σύγκρινε τη *getArr()* με τη *getCourses()* της *Student*.

Δημιουργός και καταστροφέας δεν διαφέρουν οπτικώς από αυτούς που γράψαμε στην §Prj07.1.4.

Πάμε τώρα στη

```
map< unsigned int, Student >::iterator StudentCollection::findNdx( int aIdNum)
{
   map<unsigned int,Student>::iterator ndx;
   if ( aIdNum <= 0 )
      ndx = scArr.end();
   else
      ndx = scArr.find( aIdNum );
   return ndx;
} // StudentCollection::findNdx
```

Τι δείχνει ο *ndx* και ο προσεγγιστής που επιστρέφει η *findNdx()*; Ένα ζεύγος τύπου `pair<unsigned int,Student>`.

Η *find1Student()* γίνεται όπως στην §Prj07.1.4. Πρόσεξε όμως πώς γίνεται η

```
const Student& StudentCollection::get1Student( int aIdNum ) const
{
   map<unsigned int,Student>::const_iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "get1Student",
                                   StudentCollectionXptn::notFound, aIdNum );
   return ndx->second;
} // StudentCollection::get1Student
```

Σε σχέση με αυτήν που γράψαμε στην §Prj07.1.4, δεν άλλαξε μόνον ο τύπος του *ndx* αλλά και η επιστρεφόμενη τιμή που τώρα είναι "`ndx->second`".

Ο τύπος του *ndx* αλλάζει και στην *add1Student()* αλλά εδώ αλλάζουμε και κάτι άλλο: αχρηστεύουμε την *insert1Student()*:

```
void StudentCollection::add1Student( const Student& aStudent )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aStudent.getIdNum()) );
   if ( ndx == scArr.end() ) // δεν βρέθηκε το κλειδί
   {
      if ( scPAllEnrollments == 0 )
         throw StudentCollectionXptn( "add1Student",
                                      StudentCollectionXptn::noEnroll );
```

```
      scArr.insert( make_pair(aStudent.getIdNum(),aStudent) );
      vector<Course::CourseKey> aStCourses;
      aStudent.getCourses( aStCourses );
      for ( int k(0); k < aStudent.getNoOfCourses(); ++k )
      {
         scPAllEnrollments->add1StudentInCourse(
                     StudentInCourse(aStudent.getIdNum(), aStCourses[k].s) );
      }
   }
} // StudentCollection::add1Student
```

Παρομοίως, στη *delete1Student()* αχρηστεύουμε την *erase1Student()*:

```
void StudentCollection::delete1Student( int aIdNum )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx != scArr.end() )  // υπάρχει
   {
      if ( (ndx->second).getNoOfCourses() > 0 )
         throw StudentCollectionXptn( "delete1Student",
                                      StudentCollectionXptn::enrollRef,
                                      aIdNum );
      scArr.erase( ndx );
   }
} // StudentCollection::delete1Student
```

Οι ενδιάμεσες μέθοδοι εισαγωγής/διαγραφής μαθημάτων σε έναν φοιτητή γίνονται:

```
void StudentCollection::add1Course( int aIdNum, const Course& aCourse )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "add1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   (ndx->second).add1Course( aCourse );
} // StudentCollection::add1Course

void StudentCollection::delete1Course( int aIdNum, const Course& aCourse )
{
   map<unsigned int,Student>::iterator ndx( findNdx(aIdNum) );
   if ( ndx == scArr.end() )
      throw StudentCollectionXptn( "delete1Course",
                                   StudentCollectionXptn::notFound, aIdNum );
   (ndx->second).delete1Course( aCourse );
} // StudentCollection::delete1Course
```

Τέλος, για φύλαξη/φόρτωση έχουμε:

```
void StudentCollection::swapArr( StudentCollection& rhs )
{
   scArr,swap( rhs.scArr );
} // StudentCollection::swap

void StudentCollection::save( ofstream& bout, SIndexEntry* index )
{
   if ( bout.fail() )
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::fileNotOpen );
   size_t scNOfStudents( scArr.size() );
   bout.write( reinterpret_cast<const char*>(&scNOfStudents),
               sizeof(scNOfStudents) );
   for ( map<unsigned int,Student>::const_iterator it(scArr.begin()),
         unsigned int k(0);
         it != scArr.end();  ++it, ++k )
   {
      index[k].sIdNum = it->first;
      index[k].loc = bout.tellp();
      (it->second).save( bout );
   }
   if ( bout.fail() )
```

```
      throw StudentCollectionXptn( "save",
                                   StudentCollectionXptn::cannotWrite );
} // StudentCollection::save

void StudentCollection::load( ifstream& bin )
{
   StudentCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         Student oneStudent;
         oneStudent.load( bin );
         tmp.add1Student( oneStudent );
      }
      if ( bin.fail() )
         throw StudentCollectionXptn( "load",
                                      StudentCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentCollection::load
```

Φυσικά, θα αλλάξει και η *checkWH()*:

```
void StudentCollection::checkWH( ostream& log, int maxWH ) const
{
   if ( maxWH <= 0 )
      throw StudentCollectionXptn( "checkWH",
                                   StudentCollectionXptn::negWH, maxWH );
   for ( map<unsigned int,Student>::const_iterator it(scArr.begin());
         it != scArr.end();  ++it )
   {
      if ( (it->second).getWH() > maxWH )
         log << "student with id num " << (it->second).getIdNum()
             << ": " << (it->second).getWH() << " hours/week"
             << endl;
   } // for
} // StudentCollection::checkWH
```

Prj07.2.6 Κλάση *StudentInCourseCollection*

Εδώ η απεικόνισή μας θα είναι:

```
map< StudentInCourse::SICKey, StudentInCourse > siccArr;
```

Αυτό θα πει ότι θα βάλουμε μια φορά το κλειδί και άλλη μια φορά το κλειδί μαζί με μια τιμή `float`; Θα προτιμούσες δηλαδή να βάλουμε "`map<StudentInCourse::SICKey, float>`"; Αν θέλεις, δοκίμασέ το ως μια (εύκολη) άσκηση. Να έχεις υπόψη σου πάντως ότι –σε πραγματικές συνθήκες– μπορεί να υπάρχουν και άλλα στοιχεία εκτός από έναν βαθμό.

Η παραπάνω δήλωση προϋποθέτει την επιφόρτωση του τελεστή "`<`" για την κλάση *StudentInCourse*. Αντί να κάνουμε αυτό, κυρίως για εκπαιδευτικούς λόγους, θα ορίσουμε το συναρτησοειδές:

```
   struct SICKeyLT
   {
      bool operator()( const SICKey& frst, const SICKey& scnd ) const
      {
         bool fv( false );
         if ( frst.CCode < scnd.CCode )  fv = true;
         else if ( frst.CCode == scnd.CCode )
         {
            if ( frst.sIdNum < scnd.sIdNum )  fv = true;
         }
         return fv;
      }
```

```
   }; // SICKeyLT
```

θα ορίσουμε τη συντομογραφία *SICMap* ως:

```
   typedef map< StudentInCourse::SICKey, StudentInCourse,
                StudentInCourseCollection::SICKeyLT > SICMap;
```

και θα δηλώσουμε:

```
   SICMap siccArr;
```

Πώς θα αποθηκεύονται τα αντικείμενα στην απεικόνιση; Όλα τα αντικείμενα με τον ίδιο κωδικό μαθήματος θα αποθηκεύονται μαζί (κατ' αύξοντα αριθμό μητρώου).

Πού μπαίνουν όλα αυτά; Στην αρχή της περιοχής "`private`":

```
class StudentInCourseCollection
{
public:
// . . .
private:
   struct SICKeyLT
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ
   typedef map< StudentInCourse::SICKey, StudentInCourse,
                StudentInCourseCollection::SICKeyLT > SICMap;
   SICMap siccArr;
// . . .
}; // StudentInCourseCollection
```

Σημείωση:▶

Ως άσκηση (εύκολη) δοκίμασε και τον άλλο δρόμο: επιφόρτωσε τον τελεστή "`<`" για την κλάση *StudentInCourse*:

```
bool operator<( const StudentInCourse::SICKey& frst,
                const StudentInCourse::SICKey& scnd )
{
   bool fv( false );
   if ( frst.CCode < scnd.CCode )  fv = true;
   else if ( frst.CCode == scnd.CCode )
   {
      if ( frst.sIdNum < scnd.sIdNum )  fv = true;
   }
   return fv;
} // operator<
```

Σύγκρινέ την με την επιφόρτωση του "`()`" στο συναρτησοειδές.◀

Μετά τους παραπάνω ορισμούς και δηλώσεις έχουμε δημιουργό:

```
StudentInCourseCollection::StudentInCourseCollection()
{
   siccPAllStudents = 0;
   siccPAllCourses = 0;
} // StudentInCourseCollection::StudentInCourseCollection
```

και καταστροφέα:

```
   ~StudentInCourseCollection()  { };
```

Ακόμη έχουμε:

```
   size_t getNOfStudentInCourses() const  {  return siccArr.size();  }

void StudentInCourseCollection::getArr( vector<StudentInCourse>& sicTbl) const
{
   try
   {
      for ( StudentInCourseCollection::SICMap::const_iterator
                                                    it(siccArr.begin());
            it != siccArr.end();  ++it )
         sicTbl.push_back( it->second );
   }
   catch( bad_alloc& )
   {  throw StudentCollectionXptn( "getArr",
                                   StudentCollectionXptn::allocFailed );  }
```

```
} // StudentInCourseCollection::getArr

StudentInCourseCollection::SICMap::iterator
                 StudentInCourseCollection::findNdx( int aIdNum, string code )
{
   StudentInCourseCollection::SICMap::iterator ndx;
   if ( aIdNum <= 0 || code.length() != Course::cCodeSz-1 )
      ndx = siccArr.end();
   else
      ndx = siccArr.find( StudentInCourse::SICKey(aIdNum, code) );
   return ndx;
} // StudentInCourseCollection::findNdx
```

Οι μέθοδοι για «1 *StudentInCourse*» γίνονται:

```
   bool find1StudentInCourse( int aIdNum, string code )
   {  return ( findNdx(aIdNum, code) != siccArr.end() );  }

const StudentInCourse&
           StudentInCourseCollection::get1StudentInCourse( int aIdNum,
                                                            string code ) const
{
   StudentInCourseCollection::SICMap::const_iterator
                                                 ndx( findNdx(aIdNum, code) );
   if ( ndx == siccArr.end() )
      throw StudentInCourseCollectionXptn( "get1StudentInCourse",
                                      StudentInCourseCollectionXptn::notFound,
                                      StudentInCourse::SICKey(aIdNum, code) );
   return ndx->second;
} // StudentInCourseCollection::get1StudentInCourse

void StudentInCourseCollection::delete1StudentInCourse( int aIdNum,
                                                         string code )
{
   StudentInCourseCollection::SICMap::iterator
                                       ndx( findNdx(aIdNum, code) );
   if ( ndx != siccArr.end() ) // υπάρχει στοιχείο
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn(
                                          "delete1StudentInCourse",
                             StudentInCourseCollectionXptn::noCrs );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn(
                                          "delete1StudentInCourse",
                           StudentInCourseCollectionXptn::noStdnt );
      siccPAllCourses->delete1Student( code );
      Course oneCourse( siccPAllCourses->get1Course( code) );
      siccPAllStudents->delete1Course( aIdNum, oneCourse );
      siccArr.erase( ndx );
   }
} // StudentInCourseCollection::delete1StudentInCourse

void StudentInCourseCollection::add1StudentInCourse(
                                          const StudentInCourse& aStdnInCrs )
{
   if ( findNdx( aStdnInCrs.getSIdNum(),
                           aStdnInCrs.getCCode()) == siccArr.end() )
   {
      if ( siccPAllCourses == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                             StudentInCourseCollectionXptn::noCrs );
      if ( !(siccPAllCourses->find1Course(aStdnInCrs.getCCode())))
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                          StudentInCourseCollectionXptn::unknownCrs,
                                           aStdnInCrs.getCCode() );
      if ( siccPAllStudents == 0 )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
```

```
                                        StudentInCourseCollectionXptn::noStdnt );
      if ( !(siccPAllStudents->find1Student(aStdnInCrs.getSIdNum())) )
         throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                                   StudentInCourseCollectionXptn::unknownStdn,
                                              aStdnInCrs.getSIdNum() );
      siccPAllCourses->add1Student( aStdnInCrs.getCCode() );
      Course oneCourse( siccPAllCourses->get1Course(aStdnInCrs.getCCode()) );
      siccPAllStudents->add1Course( aStdnInCrs.getSIdNum(), oneCourse );
      try
      {
        siccArr.insert( make_pair(aStdnInCrs.getKey(), aStdnInCrs));
      }
      catch( bad_alloc& )
      {  throw StudentInCourseCollectionXptn( "add1StudentInCourse",
                              StudentInCourseCollectionXptn::allocFailed );  }
   }
} // StudentInCourseCollection::add1StudentInCourse
```

Τέλος, οι *save()*, *load()* και *swapArr()* γίνονται:

```
void StudentInCourseCollection::save( ofstream& bout ) const
{
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                 StudentInCourseCollectionXptn::fileNotOpen );
   size_t siccNOfStudentInCourses( siccArr.size() );
   bout.write( reinterpret_cast<const char*>(&siccNOfStudentInCourses),
               sizeof(siccNOfStudentInCourses) );
   for ( StudentInCourseCollection::SICMap::const_iterator
                                                        it(siccArr.begin());
         it != siccArr.end();  ++it )
      (it->second).save( bout );
   if ( bout.fail() )
      throw  StudentInCourseCollectionXptn( "save",
                                 StudentInCourseCollectionXptn::cannotWrite );
} // StudentInCourseCollection::save

void StudentInCourseCollection::load( ifstream& bin )
{
   StudentInCourseCollection tmp;
   size_t n;
   bin.read( reinterpret_cast<char*>(&n), sizeof(size_t) );
   if ( !bin.eof() )
   {
      for ( int k(0); k < n && !bin.fail(); ++k )
      {
         StudentInCourse oneStudentInCourse;
         oneStudentInCourse.load( bin );
         tmp.add1StudentInCourse( oneStudentInCourse );
      }
      if ( bin.fail() )
         throw  StudentInCourseCollectionXptn( "load",
                         StudentInCourseCollectionXptn::cannotRead );
      swapArr( tmp );
   }
} // StudentInCourseCollection::load

void StudentInCourseCollection::swapArr( StudentInCourseCollection& rhs )
{
   siccArr.swap( rhs.siccArr );
} // StudentInCourseCollection::swap
```

Prj07.2.6.1 Δυο Λόγια για τη *SICKeyLT*

Είπαμε παραπάνω ότι με το συναρτησοειδές που ορίσαμε «*Όλα τα αντικείμενα με τον ίδιο κωδικό μαθήματος θα αποθηκεύονται μαζί (κατ' αύξοντα αριθμό μητρώου).*» Ας το δούμε αυτό.

Βάζουμε στο πρώτο πρόγραμμα, πριν από τη φύλαξη των συλλογών, τις εντολές:

```
log.open( "sic.txt" );
vector<StudentInCourse> allSic;
allEnrollments.getArr( allSic );
for ( int k(0); k < allSic.size(); ++k )
   allSic[k].display( log );
log.close();
```

Στο `sic.txt` βλέπουμε τα εξής:

```
2019 ΕΥ0100Ε 0
2099 ΕΥ0100Ε 0
2173 ΕΥ0100Ε 0
2069 ΕΥ01000 0
2077 ΕΥ01000 0
2081 ΕΥ01000 0
2117 ΕΥ01000 0
. . .
2067 ΤΠ03050 0
2240 ΤΠ03050 0
2044 ΤΠ03080 0
2067 ΤΠ0314Ε 0
2059 ΤΠ03140 0
2091 ΤΠ03140 0
2127 ΤΠ03140 0
2153 ΤΠ03140 0
2164 ΤΠ03140 0
2203 ΤΠ03140 0
```

ή αλλιώς:

```
ΕΥ0100Ε
2019
2099
2173
ΕΥ01000
2069
2077
2081
2117
. . .
ΤΠ03050
2067
2240
2044
ΤΠ0314Ε
2067
ΤΠ03140
2059
2091
2127
2153
2164
2203
```

Όπως βλέπεις, εδώ έχουμε τις καταστάσεις εγγεγραμμένων στα μαθήματα που δίνει η γραμματεία στους διδάσκοντες.

Καλό θα είναι να αλλάξεις τη σειρά των συγκρίσεων στο συναρτησοειδές κα να ξαναδοκιμάσεις. Μπορείς να προβλέψεις τι θα πάρεις; Τις δηλώσεις μαθημάτων ανά φοιτητή (όπως υπάρχουν στα αντικείμενα –τύπου *Student*– του μητρώου των φοιτητών).

Prj07.2.7 Πίνακας – Ευρετήριο

Στην §Prj07.2.1.5 είπαμε σχετικώς με τον πίνακα-ευρετήριο «*Δεν θα αλλάξουμε περιέχον αλλά θα ταξινομήσουμε τον πίνακα (κατ' αύξοντα αριθμό μητρώου) ώστε να υπάρχει δυνατότητα δυαδικής αναζήτησης από τα προγράμματα εφαρμογής που τον χρησιμοποιούν.*»

Ε, δεν θα χρειαστεί να κουραστούμε: ο πίνακας *είναι* ταξινομημένος όπως τον θέλουμε! Πράγματι, στην §Prj07.2.5, για την κλάση *StudentCollection*, δηλώσαμε περιέχον για τη συλλογή:

```
map< unsigned int, Student > scArr;
```

Το εννοούμενο τρίτο όρισμα του περιέχοντος είναι "`less<unsigned int>`". Στο scArr εισάγουμε στοιχεία με την

"`scArr.insert( make_pair(aStudent.getIdNum(),aStudent) )`"

και τέλος, για τη φύλαξη στο αρχείο, διασχίζουμε το περιέχον με τη:

```
for (map<unsigned int,Student>::const_iterator it(scArr.begin()),
     unsigned int k(0);
     it != scArr.end();  ++it, ++k )
{
   index[k].sIdNum = it->first;
   index[k].loc = bout.tellp();
   (it->second).save( bout );
}
```

Και πού μπαίνει η δυαδική αναζήτηση; Στο δεύτερο πρόγραμμα, αν θέλουμε. Κατ' αρχήν το δεύτερο πρόγραμμα –όπως και το πρώτο– δουλεύει μια χαρά χωρίς να χρειάζεται τροποποίηση.

Αν θέλεις να χρησιμοποιήσεις την (*std*::)*lower_bound()* για να εκμεταλλευθείς την ταξινόμηση του πίνακα θα πρέπει να κάνεις τα εξής:

- Ορίζεις την

```
bool comp( const SIndexEntry& frst, const SIndexEntry& scnd )
{  return ( frst.sIdNum < scnd.sIdNum );  } // comp
```

- Αντικαθιστάς τις:

```
int ndx( linSearch(sIndex, ndxSz, 0, ndxSz-1,
         SIndexEntry(idNum)) );
if ( ndx < 0 )
```

με τις

```
SIndexEntry* ndx( lower_bound(sIndex, sIndex+ndxSz,
                  SIndexEntry(idNum), comp) );
if ( ndx->sIdNum != idNum )
```

και τη

```
bin.seekg( sIndex[ndx].loc );
```

με τη

```
bin.seekg( ndx->loc );
```

Φυσικά, δεν θα ξεχάσεις να βάλεις και την "`#include <algorithm>`".

Prj07.3 Τι Είδαμε στα Δύο Παραδείγματα

Σύγκρινε τους δύο τρόπους που γράψαμε τις κλάσεις μας για να καταλάβεις ότι η άκριτη αποκλειστική χρήση του *vector* μπορεί να σου ευκολύνει κάπως τη δουλειά αλλά δεν σου δίνει το καλύτερο αποτέλεσμα.

Είναι ουσιώδες να κατανοήσεις καλά το πρόβλημα που έχεις να λύσεις και να επιλέξεις το καταλληλότερο περιέχον για την κάθε συλλογή δεδομένων.

Πάντως θα πρέπει να επισημάνουμε το πόσο καλό εργαλείο είναι το *vector* για την τεχνική RAII όταν ο πόρος που χειρίζεσαι είναι δυναμική μνήμη. Δηλαδή, τα άλλα περιέχοντα δεν είναι κατάλληλα; Μια χαρά είναι και τα άλλα αφού έχουν τους καταστροφείς τους αλλά το *vector* είναι το πιο απλό στον χειρισμό.

Τέλος, πρόσεξε ότι τα δικά μας εργαλεία (του `MyTmpltLib.h`) αχρηστεύθηκαν αφού η STL μας δίνει καλύτερες λύσεις.

project

8

Προβλήματα για Λύση

Περιεχόμενα:

Prj08.1 Φοιτητές και Μαθήματα: Η Απλή Λύση **1067**
Prj08.2 Αεροπλάνα και Πιλοτοι **1068**
Prj08.3 Αρχεία jpeg **1070**
Prj08.3.1 Ο Κατάλογος **1070**
Prj08.3.2 Η Σύνθεση **1071**
Prj08.4 Τραπεζικά **1072**

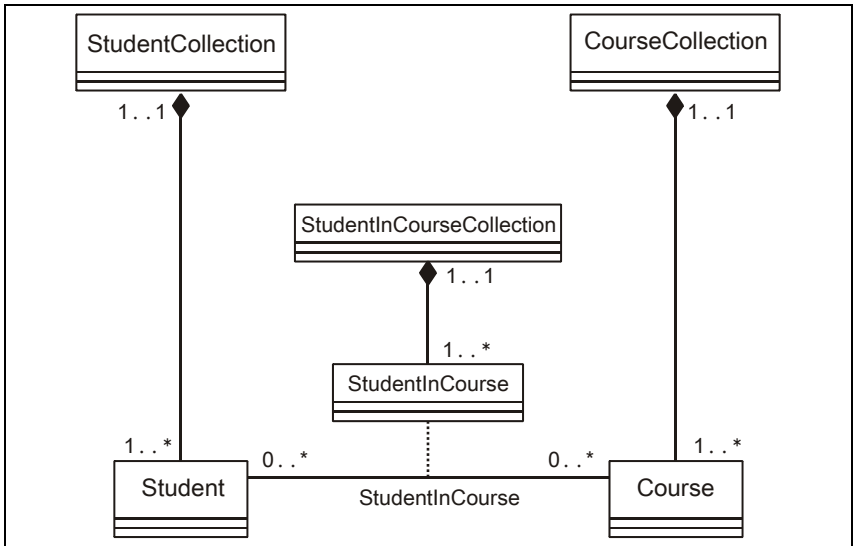
Εισαγωγικό Σημείωμα:
Στο Project αυτό δίνουμε προβλήματα για να τα λύσεις εσύ! Απόλαυσε την εργασία σου!

Prj08.1 Φοιτητές και Μαθήματα: Η Απλή Λύση

Αφού είδαμε αυτά που θέλαμε να δούμε από προγραμματισμό ας δούμε τώρα τη σωστή λύση για το πρόβλημα «Φοιτητές και Μαθήματα» που είναι και απλούστατη. Στην πραγματικότητα πρόκειται για το αρχικό σχέδιο (Project 3) αλλά με τους πίνακες ως αντικείμενα κλάσεων (*CourseCollection* κλπ). Δες το Σχ. Prj08-1.

Στην §Prj04.13 λέγαμε:

- *«Τα στοιχεία του φοιτητή επώνυμο, όνομα και αριθμός μητρώου είναι σταθερά για όλη τη διάρκεια των σπουδών του. Τα άλλα, πλήθος και κωδικοί μαθημάτων και εβδομαδιαίος*



Σχ. Prj08-1 Το διάγραμμα κλασεων της απλής λύσης.

φόρτος, έχουν σχέση με ένα συγκεκριμένο ακαδημαϊκό εξάμηνο και επαναλαμβάνονται. Για κάθε φοιτητή λοιπόν θα πρέπει να έχουμε ένα αντικείμενο με τα:

```
   unsigned int sIdNum;        // αριθμός μητρώου
   char         sSurname[sNameSz];
   char         sFirstname[sNameSz];
// άλλα στοιχεία που παραλείψαμε
```

και πολλά –ένα για κάθε ακαδημαϊκό εξάμηνο– με τα

```
   unsigned int       sIdNum;        // αριθμός μητρώου
   char               sSemester[10]; // ακαδημαϊκό εξάμηνο
   unsigned int       sWH;           // ώρες ανά εβδομάδα
   unsigned int       sNoOfCourses;  // αριθμός μαθημάτων που δήλωσε
   Course::CourseKey* sCourses;
```

- *Παρομοίως, για κάθε μάθημα θα πρέπει να έχουμε ένα αντικείμενο με τα «σταθερά» στοιχεία:*

```
   CourseKey    cCode;            // κωδικός μαθήματος
   char         cTitle[cTitleSz]; // τίτλος μαθήματος
   unsigned int cFSem;            // τυπικό εξάμηνο
   bool         cCompuls;         // υποχρεωτικό ή επιλογής
   char         cSector;          // τομέας
   char         cCateg[cCategSz]; // κατηγορία
   unsigned int cWH;              // ώρες ανά εβδομάδα
   unsigned int cUnits;           // διδακτικές μονάδες
   CourseKey    cPrereq;          // προαπαιτούμενο
```

και ένα με τα στοιχεία που αλλάζουν κάθε εξάμηνο:

```
   CourseKey    cCode;            // κωδικός μαθήματος
   unsigned int cNoOfStudents;    // αριθ. Φοιτητών
```

(και ακόμη τους αριθμούς μητρώου σπουδαστών που το παρακολουθούν, τα στοιχεία του διδάσκοντα κλπ.)»

Παίρνουμε λοιπόν το διάγραμμα κλάσεων του Σχ. Prj08-1 όπου *Student* και *Course* είναι τα σταθερά μέρη που περιγράψαμε πιο πάνω. Και τα μεταβλητά τα «πετάμε»; Ναι! Δεν μας χρειάζονται. Και όταν θέλουμε να δείξουμε τα μαθήματα που γράφηκε ένας φοιτητής; Τα βρίσκουμε από το *allEnrollments*! Οι *Student*::*getWH()*, *Student*::*getNoOfCourses()*, *Student*::*getCourses()*, *Course*::*getNoOfStudents()* θα υλοποιηθούν με μέτρημα καταχωρίσεων στο *allEnrollments*.

Λύσε λοιπόν το πρόβλημα χρησιμοποιώντας όσο πιο πολύ μπορείς την STL. Χρησιμοποίησε ό,τι θέλεις από τις λύσεις που δώσαμε πιο πριν αλλά να θυμάσαι ότι σε κάποιες περιπτώσεις, για να δείξουμε ορισμένα πράγματα, δεν χρησιμοποιήσαμε τις απλούστρες επιλογές.

Prj08.2 Αεροπλάνα και Πιλοτοι

Το πληροφοριακό σύστημα της Karakaxa Airlines (KA) περιλαμβάνει πληροφορίες για προσωπικό, αεροπλάνα, πτήσεις και δρομολόγια.

Για κάθε πιλότο έχουμε αντικείμενο της μορφής:

```
class Pilot
{
public:
// . . .
private:
   unsigned int   piIdNum;         // αριθμός μητρώου
   char           piSurname[20];   // επώνυμο
   char           piFirstName[16]; // όνομα
   ?????          piExper;         // εμπειρία σε αεροσκάφη
}; // Pilot
```

όπου *piExper* κάποιο περιέχον με στοιχεία τύπου[1]

```
struct ExperAircraft
{
   char   xaAircraftType[20]; // τύπος αεροσκάφους
   double xaFlightTime;       // ώρες πτήσης
}; // ExperAircraft
```

Ιδιαίτερο ενδιαφέρον έχει η εμπειρία του κάθε πιλότου σε διάφορους τύπους αεροσκαφών. Κάθε στοιχείο της *piExper* μας λέει πόσες ώρες πτήσης (`xaFlightTime`) έχει ο πιλότος στον συγκεκριμένο τύπο αεροσκάφους (`xaAircraftType`).

Εφοδίασε την κλάση *Pilot* με μεθόδους *save()* και *load()* που φυλάγουν τα δεδομένα ενός πιλότου σε ένα μη-μορφοποιημένο (binary) αρχείο και το ξαναφορτώνουν από αυτό.

Συμπλήρωσε την κλάση *Pilot* με ό,τι (μέλος, μέθοδο) πιστεύεις ότι χρειάζεται (μέλη μπορείς να προσθέσεις αλλά όχι να αφαιρέσεις).

Το αρχείο text `pilots.txt` έχει γραμμές σαν και τις παρακάτω:

```
2105\tΚΡΗΤΙΚΟΣ\tΕΛΕΥΘΕΡΙΟΣ\n
2088\tΚΥΛΙΔΗΣ\tΔΗΜΟΣΘΕΝΗΣ\n
```

(πιλότος με αριθμό μητρώου 2105, επώνυμο ΚΡΗΤΙΚΟΣ και όνομα ΕΛΕΥΘΕΡΙΟΣ· πιλότος με αριθμό μητρώου 2088, επώνυμο ΚΥΛΙΔΗΣ και όνομα ΔΗΜΟΣΘΕΝΗΣ)

Το πρώτο πρόγραμμα που θα γράψεις θα διαβάζει το αρχείο `pilots.txt` και θα αποθηκεύει τα στοιχεία όλων των πιλότων σε περιέχον

```
????? allPilots;
```

Για κάθε αεροπλάνο έχουμε αντικείμενο της μορφής:

```
class AircraftType
{
public:
// . . .
private:
   char         atType[20];        // τύπος αεροσκάφους
   unsigned int atNoOfSeats;       // αριθμός θέσεων επιβατών
   double       atMaxTakeOffLoad;  // μέγιστο βάρος απογείωσης σε kgr
   ???????      atExperPilots;     // πιλότοι με εμπειρία
}; // AircraftType
```

όπου *atExperPilots* περιέχον με στοιχεία τύπου[2]

```
struct ExperPilot
{
   unsigned int xpIdNum;      // αρθμός μητρώου πιλότου
   double       xpFlightTime; // ώρες πτήσης
}; // ExperPilot
```

Ιδιαίτερο ενδιαφέρον έχουν οι πιλότοι της εταιρείας που μπορούν να πετάξουν τον συγκεκριμένο τύπο αεροσκάφους. Αυτοί συνοψίζονται στο περιέχον *atExperPilots*. Κάθε στοιχείο του μας λέει πόσες ώρες πτήσης (`xpFlightTime`) έχει στον συγκεκριμένο τύπο αεροσκάφους ο πιλότος με αριθμό μητρώου `xpIdNum`.

Εφοδίασε την κλάση *AircraftType* με μεθόδους *save()* και *load()* που φυλάγουν τα δεδομένα ενός αεροσκάφους σε μη-μορφοποιημένο αρχείο και το ξαναφορτώνουν από αυτό.

Το αρχείο text `aircraftTypes.txt` έχει γραμμές σαν και τις παρακάτω:

```
AIRBUS A340-300\t295\t275000\n
ATR - 42 – 320\t50\t16700 \n
```

(αεροπλάνο τύπου AIRBUS A340-300, με 295 θέσεις επιβατών και μέγιστο βάρος απόγείωσης 275000 kgr.)

Μετά τους πιλότους, το πρώτο πρόγραμμα θα διαβάζει το αρχείο `aircraftTypes.txt` και θα αποθηκεύει τα στοιχεία όλων των πιλότων σε περιέχον

---

[1] Καλύτερο θα ήταν να δηλώσουμε (`public`) την *ExperAircraft* μέσα στην κλάση.

[2] Καλύτερο θα ήταν να δηλώσουμε (`public`) την *ExperPilot* μέσα στην κλάση.

```
?????? allAircraftTypes;
```

Στη συνέχεια θα διαβάζει το αρχείο `flightHours.txt` και θα συμπληρώνει όπου χρειάζεται τα στοιχεία των πιλότων (θα γεμίζει τα περιέχοντα *piExper* των αντικειμένων που παριστάνουν τους πιλότους).

Το αρχείο text `flightHours.txt` έχει γραμμές σαν και τις παρακάτω:

```
2040\tAIRBUS A319\t21.7\n
2145\tATR - 42 - 320\t1374.2\n
```

που σημαίνει ότι ο πιλότος με αριθμό μητρώου 2040, έχει 21.7 ώρες πτήσης με αεροπλάνα τύπου AIRBUS A319, ενώ ο πιλότος με αριθμό μητρώου 2145, έχει 1374.2 ώρες πτήσης με αεροπλάνα τύπου ATR - 42 – 320.

Τέλος, θα φυλάγει σε μη-μορφοποιημένα αρχεία

- τα ενημερωμένα στοιχεία των πιλότων στο αρχείο με όνομα `pilotExp.dta` και
- τα ενημερωμένα στοιχεία των τύπων αεροσκαφών στο αρχείο `aircraftTypes.dta`.

Ένα δεύτερο πρόγραμμα θα φορτώνει καταλλήλως τα περιεχόμενα των `pilotExp.dta` και `aircraftTypes.dta` και θα ενημερώνει τα περιεχόμενά τους από τα δεδομένα των πτήσεων (που ολοκληρώνονται) που θα διαβάζει από το πληκτρολόγιο.

Για να απλουστεύσουμε τα πράγματα, για κάθε πτήση θα διαβάζουμε μόνον:

- διάρκεια πτήσης (πρώτα λεπτά),
- τύπος αεροσκάφους (πρέπει να υπάρχει στο *allAircraftTypes*),
- αριθμός μητρώου κυβερνήτη (πρέπει να υπάρχει στο *allPilots*),
- αριθμός μητρώου συγκυβερνήτη (πρέπει να υπάρχει στο *allPilots*).[3]

Προσοχή! Για να γίνουν δεκτά τα στοιχεία πτήσης θα πρέπει: κυβερνήτης και συγκυβερνήτης να έχουν τον τύπο αεροσκάφους της πτήσης σε αυτά «που ξέρουν» (έστω και με 0 ώρες). Δηλαδή, η διαδικασία ενημέρωσης του συστήματος ότι «*εκπαιδεύτηκα και σε άλλο αεροπλάνο*» δεν είναι αυτόματη.

Αν θέλεις δώσε (στο δεύτερο πρόγραμμα) τη δυνατότητα να δείχνει όλα τα στοιχεία ενός πιλότου ή/και ενός τύπου αεροσκάφους.

Παρατηρήσεις:▶

1. Λύσε το πρόβλημα χρησιμοποιώντας όσο πιο πολύ μπορείς την STL.
2. Προσοχή στον πλεονασμό, διότι υπάρχει αρκετός: Τα ίδια δεδομένα υπάρχουν στους πιλότους και στους τύπους αεροσκαφών.
3. Θέλεις να δοκιμάσεις κάτι πιο απλό (σαν την Prj08.1); Δοκίμασέ το, αλλά προσοχή: Δεν μπορείς να αφαιρέσεις από τα στοιχεία του πιλότου τους τύπους αεροσκαφών που μπορεί να κυβερνήσει.◀

Prj08.3 Αρχεία jpeg

Prj08.3.1 Ο Κατάλογος

Μια εταιρεία πουλάει διάφορα προϊόντα και θέλει να κάνει έναν ψηφιακό κατάλογο προϊόντων για τους πελάτες της. Κάθε προϊόν θα αποθηκεύεται ως αντικείμενο κλάσης

```
class Product
{
public:
// . . .
private:
```

[3] Κυβερνήτης και συγκυβερνήτης αυξάνουν κατά «διάρκεια πτήσης» τις ώρες πτήσης που έχουν με τον τύπο αεροσκάφους.

```
   char           prCode[20];     // κωδικός προϊόντος
   char           prDescr[100];   // περιγραφή
   double         prL, prW, prH;  // Διαστάσεις (μήκος, πλάτος, ύψος) σε cm
   unsigned char* prJpeg;         // φωτογραφία σε jpeg
   unsigned int   prJpegSz;       // μεγεθος του jpeg σε bytes
   char           prJpegPath[130];// πλήρης διαδρομή και όνομα αρχείου jpeg
   void displayJpeg( ostream& tout,
                     const unsigned char* jpegImg,
                     int jpegSz ) { };
}; // Product
```

Ειδικώς για τα *prJpeg*, *prJpegSz*, *prJpegPath* είναι δεκτές οι εξής καταστάσεις:

- *prJpeg* `!=` 0 (`NULL`), δηλαδή υπάρχει φωτογραφία, *prJpegSz* > 0 (μας δίνει το μέγεθος της φωτογραφίας σε bytes), *prJpegPath* έχει ως τιμή το όνομα αρχείου (με πλήρη διαδρομή) όπου θα αποθηκευτεί η φωτογραφία με το επόμενο μήνυμα *save*.
- *prJpeg* == 0 (δεν υπάρχει φωτογραφία), *prJpegSz* == 0 και *prJpegPath* `== ""`.

α) Γράψε όλες τις μεθόδους που νομίζεις ότι χρειάζεται η κλάση.

β) Πέρα από όποιες άλλες μεθόδους θεωρήσεις απαραίτητες, η κλάση θα πρέπει να έχει και τις εξής:

```
   void save( ostream& bout ) const;
```

που θα φυλάγει τις τιμές μελών ενός αντικειμένου κλάσης *Product* σε ένα μη-μορφοποιημένο αρχείο συνδεμένο με το ρεύμα *bout*. Προσοχή όμως: η φωτογραφία (αν υπάρχει) δεν φυλάγεται μέσω του ρεύματος *bout* αλλά σε άλλο αρχείο (binary) του οποίου το όνομα (με πλήρη διαδρομή) υπάρχει στο μελος *prJpegPath*.

```
   void load( istream& bin );
```

που θα φορτώνει τις τιμές μελών ενός αντικειμένου κλάσης *Product* από ένα αρχείο binary συνδεμένο με το ρεύμα *bout*. Μετά τη φόρτωση τα *prJpeg*, *prJpegSz* θεωρείται ότι έχουν τιμή 0. Αν *prJpegPath* `!= ""` τότε φορτώνεται και η φωτογραφία από το αρχείο που το όνομά του υπάρχει στο *prJpegPath*.

```
   void loadJpeg( string jPath );
```

που τροφοδοτείται με το όνομα αρχείου (με πλήρη διαδρομή) που περιέχει την εικόνα και τη φορτώνει σε δυναμικό πίνακα που δείχνει η *prJpeg*. Αν η φόρτωση γίνει επιτυχώς ενημερώνονται καταλλήλως και τα *prJpegSz*, *prJpegPath*. Αν η φόρτωση αποτύχει παραμένουν τα πάντα όπως ήταν.

Όπως φαίνεται, χειριζόμαστε μια εικόνα jpeg σαν μια ακολουθία bytes. Όταν δίνουμε *save* το αντικείμενο θα αποθηκευτεί στο αρχείο που είναι συνδεμένο με το ρεύμα *bout* αλλά η εικόνα θα αποθηκευτεί σε ξεχωριστό αρχείο που καθορίζεται στο *prJpegPath*.

γ) Στο αρχείο `products.dta` (binary) υπάρχουν μερικά προϊόντα σε αντικείμενα της παραπάνω κλάσης. Γράψε μέθοδο *display* που θα δείχνει όλα τα στοιχεία ενός προϊόντος σε ένα αρχείο text. Όλα; Και την εικόνα; Καλά... Για την εικόνα θα καλείς τη βοηθητική συνάρτηση *displayJpeg* που είναι «άδεια» (`{ }`)!

Χρησιμοποίησε

- τη *load()* για να διαβάσεις τα στοιχεία από το αρχείο και
- τη *display()* για να βγάλεις τα στοιχεία όλων των προϊόντων στην οθόνη.

δ) Σε κάποιο προϊόν βάλαμε λάθος εικόνα. Βάλαμε ως αρχείο εικόνας το `rondinejr.jpg` αντί για `Product10.jpg`. Βρες το προϊόν και διόρθωσέ το με χρήση της *loadJpeg()*. Στη συνέχεια, χρησιμοποιώντας τη *save()* φύλαξε όλα τα (σωστά) στοιχεία.

Prj08.3.2 Η Σύνθεση

Στην §Prj08.3.1 είδαμε μια κλάση χρήσιμη σε μια εταιρεία για τον ψηφιακό κατάλογο των προϊόντων της. Η εταιρεία θέλει και μια κλάση

```
class ProductConstr
{
// . . .
}; // ProductConstr
```

για την κατασκευή των προϊόντων που παράγει η ίδια.

Η *ProductConstr* έχει τα χαρακτηριστικά και τη συμπεριφορά (μεθόδους) της *Product* και περιέχει επιπλέον όλα τα εξαρτήματα (συνιστώσες) που απαιτούνται για την κατασκευή μιας μονάδας του προϊόντος. Τα εξαρτήματα περιγράφονται σε ζεύγη (κωδικός συνιστώσας, ποσότητα), π.χ. (CRW0756, 6), (CXL0718, 10).

Με βάση την *Product* γράψε την κλάση *ProductConstr*. Η νέα κλάση θα έχει όλες τις ιδιότητες της *Product*. Φυσικά οι *save*, *load* και *display* θα αλλάξουν και επιπλέον: Σε έναν (δυναμικό) πίνακα του κάθε αντικειμένου κρατούμε τα ζεύγη (κωδικός συνιστώσας, ποσότητα) του προϊόντος. Θα χρειαστούμε και μεθόδους για τη διαχείριση του πίνακα.

Στο αρχείο `products.dta` (το ξέρεις από την προηγούμενη άσκηση) υπάρχουν τα στοιχεία των προϊόντων (αντικείμενα κλάσης *Product*). Στο αρχείο text `prodComp.txt` υπάρχουν τα στοιχεία σύνθεσης των προϊόντων. Σε κάθε γραμμή υπάρχουν τριάδες:

Κωδικός προϊόντος `tab` κωδικός εξαρτήματος `tab` ποσότητα

Γράψε πρόγραμμα που θα διαβάζει τα δύο αρχεία και θα δημιουργεί τα αντικείμενα κλάσης *ProductConstr*, ένα για κάθε προϊόν. Προσοχή! Το δεύτερο αρχείο δεν έχει ελεγχθεί. Μπορεί λοιπόν να βρεις, ας πούμε, μια γραμμή

```
LED-R  NQE4868      9
```

και παρακάτω:

```
LED-R  NQE4868      8
```

Στη συνέχεια θα δίνει τη δυνατότητα στον χρήστη να κάνει διορθώσεις. Δηλαδή θα δείχνει τη σύνθεση του προϊόντος που ζητάει ο χρήστης και θα τουδίνει τη δυνατότητα α) να σβήσει κάποιο εξάρτημα β) να εισαγάγει ένα νέο εξάρτημα με την αντίστοιχη ποσότητα γ) να αλλάξει την ποσότητα κάποιου εξαρτήματος που ήδη υπάρχει.

Τέλος, θα φυλάγει όλα τα αντικείμενα σε ένα αρχείο binary δημιουργώντας και το κατάλληλο ευρετήριο για να είναι δυνατή η ανάκτηση.

Prj08.4 Τραπεζικά

Μια τράπεζα για να μηχανοργανώσει τις εργασίες της χρειάζεται κάποιες κλάσεις. Ας δούμε μερικές από αυτές.

Κλάση *Person* που τα αντικείμενά της έχουν τα στοιχεία ενός ανθρώπου που έχει (ή είχε στο παρελθόν) κάποια σχέση με την τράπεζα:

- ΑΔΤ, Αριθμός Δελτίου Ταυτότητας (δύο κεφαλαία ελληνικά γράμματα και έξη δεκαδικά ψηφία από τα οποία το πρώτο δεν είναι 0, π.χ. `KM209084`),
- επώνυμο,
- όνομα,
- (ένας) αρ. τηλεφώνου (10ψήφιος).

Στο αρχείο `persons.txt` υπάρχουν τέτοια στοιχεία, μια γραμμή για κάθε άνθρωπο, πχ.:

```
KM209084\tΓΑΡΥΦΑΛΛΟΣ\tΞΕΝΟΦΩΝ\t6997208 489\n
```

Κλάση *Loan* που τα αντικείμενά της έχουν στοιχεία δανείων:

- κωδ. δανείου (αποτελείται από δύο πενταψήφιους ορμαθούς, π.χ.: `22555 70105`), είδος δανείου (στεγαστικό, επαγγελματικό κλπ) ακέραιος από 1 μέχρι 6,
- ΑΔΤ (του μοναδικού) δανειολήπτη,
- διάρκεια σε μήνες,

- ποσό.

Στο αρχείο `loans.txt` υπάρχουν γραμμές με τέτοια στοιχεία, π.χ.:

```
22555 70105\t5\t467\t190321
```

που σημαίνει: κωδ. δανείου: `22555 70105`, τύπος δανείου `5` (ας πούμε στεγαστικό), διάρκεια δανείου `467` μήνες (κάπου 39 χρόνια), συνολικό ποσό `190321`€.

Κλάση *Account* που τα αντικείμενά της είναι στοιχεία λογαριασμών:

- κωδικός IBAN[4] (για την Ελλάδα αποτελείται 27 χαρακτήρες οι δύο πρώτοι είναι GR, οι δύο επόμενοι είναι ψηφία ελέγχου και οι υπόλοιποι (Basic Bank Account Number) έχουν σχέση με την τράπεζα, το υποκατάστημα, τον λογαριασμό),
- είδος λογαριασμού (ταμιευτηρίου, όψεως κλπ) ακέραιος από 1 μέχρι 5,
- υπόλοιπο,
- ΑΔΤ όλων των δικαιούχων του λογαριασμού.

Στο αρχείο `accounts.txt` υπάρχουν γραμμές με τέτοια στοιχεία[5], π.χ.:

```
GR0133444402010136120121721\t4\t14110
```

που σημαίνει: IBAN λογαριασμού: `GR0133444402010136120121721`, τύπος λογαριασμού `4` (ας πούμε ταμιευτήριο), υπόλοιπο `14110`€.

Μια κλάση *Customer* που θα κληρονομεί την *Person* και θα έχει επί πλέον τους κωδικούς όλων των δανείων από την και τους IBAN όλων των λογαριασμών στην τράπεζα του συγκεκριμένου ανθρώπου (πελάτη).

Το αρχείο `customers.txt` έχει γραμμές όπως:

```
Λ\tGR2333407023003420332160230\tTY212003
Δ\t50281 33128\tOP215070
```

Η πρώτη σημαίνει: ο λογαριασμός (`Λ`): `GR2333407023003420332160230`, έχει δικαιούχο τον πελάτη με ΑΔΤ `TY212003` (μπορεί να έχει και άλλους). Η δεύτερη σημαίνει ότι το δάνειο (`Δ`) `50281 33128` έχει (μοναδικό) δανειολήπτη τον πελάτη με ΑΔΤ `OP215070`.

Α) Υλοποίησε τις κλάσεις.

Β) Γράψε πρόγραμμα που θα διαβάζει τα αρχεία και θα αποθηκεύει καταλλήλως τα περιεχόμενά τους. Προσοχή! Κατά την ανάγνωση να γίνονται έλεγχοι: τα αρχεία είναι text και δεν είναι ελεγμένα.

Γ) Το πρόγραμμα θα φυλάγει καταλλήλως τα δεδομένα σε αρχεία binary, ώστε να είναι δυνατή η (σωστή) επαναφόρτωσή τους.

---

[4] International Bank Account Number. Για περισσότερα διάβασε το «*Ερωτήσεις και Απαντήσεις για τον IBAN*» της Ένωσης Ελληνικών Τραπεζών (http://www.hba.gr/iban/IBAN-gr.pdf).

[5] Τα ψηφία ελέγχου στους IBAN του αρχείου είναι λάθος (βαλμένα στην τύχη). Όποιος/α θέλει ας ψάξει στο διαδίκτυο να βρει πώς υπολογίζονται με τη μέθοδο MOD 97-10 και ας τα διορθώσει.

βιβλιογραφία

Το (Stroustrup 1997) είναι το πρώτο βιβλίο που θα δούμε. Σίγουρα δεν θα ξεκινήσεις να μαθαίνεις προγραμματισμό και C++ από αυτό. Αλλά, εκτός του ότι περιέχει ολόκληρο το πρότυπο της C++ είναι πολύτιμο διότι περιλαμβάνει:

- μια εξαιρετική παρουσίαση της γλώσσας από τον δημιουργό της,
- συμβουλές για τη χρήση (καλύτερα την αξιοποίηση) των χαρακτηριστικών της,
- συμβουλές για καλό προγραμματισμό από έναν καλό προγραμματιστή.

Η προηγούμενη (2η) έκδοση έχει μεταφραστεί στα ελληνικά.

Πριν δούμε άλλα βιβλία για τη C++ θα πρέπει να σταθούμε σε δύο βιβλία για τη C. Το πρώτο, (Kernighan & Ritchie 1988), είναι πιθανότατα το πιο πολυδιαβασμένο βιβλίο προγραμματισμού διεθνώς. Το δεύτερο, (Kernighan 1996), συνεχίζει την επιτυχία του προηγούμενου. Μπορεί τα βιβλία να είναι για C αλλά, διαβάζοντάς τα, θα μάθεις πολλά πολύτιμα πράγματα για προγραμματισμό *και για C++*. Το πρώτο έχει μεταφραστεί και στα ελληνικά.

Το (Lippman & Lajoie 2013) ήταν, για χρόνια, το πιο διαδεδομένο διδακτικό εγχειρίδιο όπου διδάσκεται C++. Η 5η έκδοση καλύπτει και το C++11. Είναι εξαιρετικό αλλά δεν θα ξεκινήσεις από αυτό.

Πιο εύκολο για ξεκίνημα είναι το (Deitel & Deitel 2013)· η 6η έκδοση έχει μεταφραστεί στα ελληνικά (Μ. Γκιούρδας), η 9η έκδοση καλύπτει και το C++11 και η 8η (αγγλικά) υπάρχει δωρεάν στο διαδίκτυο.[1] Η Deitel προσφέρει δωρεάν στο διαδίκτυο πολλά υλικά καθώς και μαθήματα.[2]

Να αναφέρουμε και τα (Jamsa 1994), (Liberty 1995), (Lafore 2001)· οι εκδόσεις τους στα αγγλικά ήταν best-sellers και υπάρχουν ελληνικές μεταφράσεις. Στα ελληνικά είναι και το (Σταμούλης 1992).

Αν ξέρεις αγγλικά οι επιλογές σου είναι πολύ περισσότερες:

Τα (Hanly 1997) και (Mansfield & Antonakos 1997) είναι εύκολα, εισαγωγικά.

Τα best-sellers της εποχής είναι: (Schildt 1997), (Schildt 1998), (Satir & Brown 1995), (Yaroshenko 1995). Αλλά προσοχή: όλα, εκτός από το τελευταίο, απευθύνονται σε ανθρώπους που ξέρουν ήδη προγραμματισμό ή –ακόμη περισσότερο– ξέρουν ήδη C.

Το (Feuer 1982) είναι ένα βιβλίο με ασκήσεις στη C (*όχι C++*), που κάποτε «πούλησε» πολύ.

Για όσους έχουν πιο συγκεκριμένα ενδιαφέροντα:

Για περισσότερα στη *Μαθηματική Λογική* σε παραπέμπουμε στα (Τζουβάρας 1987**), (**Κυριακόπουλος 1977**), (**Manna & Waldinger 1985).

Για γλώσσες προδιαγραφών: (Andrews & Ince 1991), (Diller 1990), (Jones 1990), (Spivey 1988).

Θα πρέπει να επιστήσουμε την προσοχή σου σε τρεις αναφορές σε διαδικτυακούς τόπους:

- (ELLEMTEL, 1992) «*Programming in C++: Rules and Recommendations*»: Πρόκειται για προγραμματιστικούς κανόνες που πρέπει να ακολουθούνται από τους προγραμματιστές

[1] http://aicitel.files.wordpress.com/2011/12/deitel_2012.pdf

[2] http://www.deitel.com/FreeTutorials/FreeTutorialIndex/tabid/1575/Default.aspx#CPLUSPLUS

της ELLEMTEL, μιας εταιρείας ανάπτυξης ηλεκτρονικών συστημάτων για την Ericsson και την Televerket.

- (Lockheed Martin, 2005) «*Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*»: Έχει κανόνες ανάπτυξης λογισμικού σε C++ για ένα σύστημα με κρισιμότητα ασφάλειας (safety critical), το σύστημα ενός νέου μαχητικού αεροπλάνου.
- (CERT 2009) «*CERT C++ Secure Coding Standard*»: Πρόκειται για μια έκδοση του Software Engineering Institute (SEI) με κανόνες για να γράφεις ασφαλή προγράμματα σε C++. Δεν είναι περίεργο ότι έχει αρκετές αναφορές στους δύο προηγούμενους ιστότοπους.

Πέρα από τα παραπάνω, στο διαδίκτυο θα βρεις πολλούς τόπους με ερωταποκρίσεις σχετικώς με τη C++ και τον προγραμματισμό.[3] Να επισημάνουμε

- Τον «κλασικό» (Cline 1999)· μπορείς να βρεις και την έντυπη μορφή του (Cline, Lomow, Girou 2000).
- Τον «Guru of the Week» (Sutter 1998) που εκδόθηκε ως *Exceptional C++* (Sutter 1999) που υπάρχει και ως pdf.

Για τις διαφορές του C++11 (και του C++14) από το C++03 παραπέμπουμε κατ' αρχάς στο (Stroustrup, B. 2011). Δες ακόμη το (Kalev 2011).

Αν σε ενδιαφέρουν τα πρότυπα θα βρεις:

- Το C++03 στο (ISO 2003).
- Το C++11 στο (ISO 2011). Για να το πάρεις θα πρέπει να πληρώσεις.
- Στο (ISO/IEC JTC1 SC22 WG21 N3337 2012) θα βρεις δωρεάν το έγγραφο εργασίας που είναι σχεδόν ίδιο με το C++11.
- Στο (ISO/IEC JTC1 SC22 WG21 N3797 2014) θα βρεις το έγγραφο εργασίας που είναι σχεδόν ίδιο με το C++14.

Θα πρέπει να τονίσουμε πάντως ότι τα πρότυπα δεν θα σε βοηθήσουν και πολύ.

Κάππος, Δ. 1962: *Μαθήματα Αναλύσεως - Απειροστικός Λογισμός* τ. Α', Αθήνα.

Κατσαργύρη, Β., Κ. Μεντή, Γ. Παντελίδη, Κ. Σούρλα, 1992: *Μαθηματικά Γ' Λυκείου, Ανάλυση*, Ο.Ε.Δ.Β., Αθήνα.

Κυριακόπουλου, Α. Κ. 1977: *Μαθηματική Λογική μετά Μεθόδων Αποδείξεως εις τα Μαθηματικά*, 2η Έκδ., Παπαδημητρόπουλος, Αθήνα.

Σταμούλη, Α. 1992: *Το Βιβλίο της C++*, Learning Plan, Θεσσαλονίκη.

Τζουβάρα, Α. 1987: *Στοιχεία Μαθηματικής Λογικής*, Θεσσαλονίκη.

Abrahams, D. 2001: *Exception-Safety in Generic Components*, retrieved on 28.12.2012 from: `http://www.boost.org/community/exception_safety.html`

Abrahams, D. 2010: *Error and Exception Handling*, retrieved on 28.12.2012 from: `http://www.boost.org/community/error_handling.html`

Alexandrescu, A. 2001: *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley, Reading Ma, US.

Andrews, D. & D. Ince 1991: *Practical Formal Methods with VDM*, McGraw-Hill, London, UK.

Bohm, C. & G. Jacopini 1966: "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *CACM* v.**9**, pp. 366-71.

CERT, 2009: *CERT C++ Secure Coding Standard*, retrieved on 01.06.2009 from: `http://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637`.

[3] Γενικώς –και όχι μόνον για προγραμματισμό– θα πρέπει να υπενθυμίσουμε ότι υπάρχουν ΔΩΡΕΑΝ τα υλικά του OpenCourseWare (OCW) από τα μεγαλύτερα πανεπιστήμια του κόσμου! Στα αγγλικά; Ναι, στα αγγλικά…

Cline, M. 1999: *C++ FAQ*, retrieved on 05.03.2007 from: `http://www.parashift.com/c++-faq-lite/index.html`.

Cline M., G. Lomow, M. Girou 2000: *C++ FAQs* (Second Edition), Addison Wesley, 2000, retrieved on 19.08.2014 from: `http://mmc.geofisica.unam.mx/femp/Herramientas/CyC++/Documentacion/C++FAQs.pdf`.

Deitel, H.M. & P.J. Deitel 2013: *C++ How to Program* 9th ed., Prentice-Hall, Upper Saddle River, NJ, USA.

Dijkstra, E.W. 1988: "Go to Statement Considered Harmful", *CACM* v.**11**, pp. 147-148.

Diller, A. 1990: *Z: An Introduction to Formal Methods*, J. Wiley & Sons, Chichester, UK.

ELLEMTEL, 1992: *Programming in C++: Rules and Recommendations*, Document: M 90 0118 Uen, Rev. C, 27 April 1992, retrieved on 01.06.2009 from: `http://www.doc.ic.ac.uk/lab/cplus/c++.rules/`

Feuer, A.R. 1982: *The C Puzzle Book*, Prentice Hall, Englewood Cliffs, NJ, US.

Haendel, L. 2001: *www.Function-Pointer.org: The site dedicated to C and C++ Function Pointers*, retrieved on Oct 2009 from: `http://www.newty.de/fpt/index.html`

Hanly, J.R. 1997: *Essential C++ for Engineers and Scientists*, Addison-Wesley, Reading Ma, US.

Henricson, M. & E. Nyquist 1997: *Industrial Strength C++ - Rules and Recommendations*, Prentice Hall, Upper Saddle River, NJ, US.

Hosey, P. 2007: *Everything you need to know about pointers in C*, retrieved on 10.10.2008 from: `http://boredzo.org/pointers/`

ISO/IEC 1999 (C99): *Programming Languages — C, Second Edition*, ISO/IEC 9899-1999.

ISO 2003: *International Standard ISO-IEC 14882: Programming Language C++*, Second Edition, 2003-10-15.

ISO 2011: *International Standard ISO-IEC 14882: Information technology — Programming languages — C++*, Third Edition, 2011-09-01.

ISO/IEC JTC1 SC22 WG21 N3337 2012 (C++0X): *Programming Languages — C++* (Working Draft Standard), ISO 2012, retrieved on 16.05.2012 from: `http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`

ISO/IEC JTC1 SC22 WG21 N3797 2014 (C++14): *Programming Languages — C++* (Working Draft Standard), ISO 2013, retrieved on 29.05.2014 from: `http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf`

Jamsa, K. 1994: *Εισαγωγή στη C++*, Κλειδάριθμος, Αθήνα.

Jones, C.B. 1990: *Systematic Software Development using VDM*, Prentice Hall, London, UK.

Kalev, D. 2011: *The Biggest Changes in C++11 (and Why You Should Care)*, retrieved on 19.08.2014 from: `http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/`.

Kernighan, B.W. & R. Pike 1996: *The Practice of Programming*, Addison-Wesley, Reading Ma, US.

Kernighan, B.W. & D.M. Ritchie 1988: *The C Programming Language*, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, US.

Knuth, D. 1977: "Structured Programming with **goto** Statements", στο *Current Trends in Programming Methodology*, R.T. Yeh (ed.) vol. 1, Prentice Hall, Englewood Cliffs, NJ, US, αναδημοσίευση από το *Comp. Surveys*, **6**, 4, pp. 261-301 (1974).

Lafore, R. 2001: *Αντικειμενοστρεφής προγραμματισμός με τη C++*, Κλειδάριθμος, Αθήνα.

Liberty, J. 1995: *Εγχειρίδιο C++*, Γκιούρδας, Αθήνα.

Lippman, S.B., J.Lajoie, B.E.Moo 2013: *C++ Primer*, 5th ed., Addison-Wesley, Reading Ma, US. Retrieved on 19.08.2014 from: http://dl.e-book-free.com/2013/07/ c_primer_5th_edition.pdf.

Liskov, B. & J. Wing 1993: *Family Values: A semantic notion of subtyping*, MIT Lab for Computer Science TR-562b, retrieved on 10.10.2001 from: http://citeseer.ist.psu.edu/liskov94family.html.

Lockheed Martin, 2005: *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, doc. num.: 2RDU00001 Rev C., Dec 2005, retrieved on 01.06.2009 from: http://www.research.att.com /%7Ebs/JSF-AV-rules.pdf.

Manna, Z. & R. Waldinger 1985: *The Logical Basis for Computer Programming, Vol.1: Deductive Reasoning*, Addison-Wesley, Reading Ma, US.

Mansfield, K.C.Jr & J.L.Antonakos 1997: *An Introduction to Programming using C++*, Prentice Hall, Englewood Cliffs, NJ.

OMG 2010: *OMG Unified Modeling Language™ (OMG UML), Superstructure,* Ver. 2.3, May 2010, retrieved on 19.12.2010 from: http://www.omg.org/spec/UML/2.3/Superstructure.

Satir, G. & D. Brown 1995: *C++ : The Core Language*, O'Reilly & Associates, New York, NY, US.

Schildt, H. 1997: *Teach Yourself C++*, Osborne McGraw-Hill, New York, NY, US.

Schildt, H. 1998: *C++ from the Ground Up*, Osborne McGraw-Hill, New York, NY, US.

SGI 1999: *Standard Template Library Programmer's Guide*, retrieved on 02.06.2003 from http://www.sgi.com/tech/stl/

Spivey, J.M. 1988 : *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge Univ. Press, Cambridge, UK.

Stepanov, A. A., 2007: *Notes on Programming*, retrieved on 12.08.2008 from http://www.stepanovpapers.com/

Stroustrup, B. 1997: *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading Ma, US.

Stroustrup, B. 2011: *C++11 - the new ISO C++ standard*, retrieved on 19.08.2014 from http://www.stroustrup.com/C++11FAQ.html

Sutter, H. 1998: *GotW #49: Template Specialization and Overloading*, Dec 2008, retrieved on March 2007 from: http://www.gotw.ca/gotw/049.htm.

Sutter, H. 1999: *Exceptional C++: 47 Engineering Puzzles*, Addison Wesley, 1999, retrieved on 20.08.2014 from http://aalmos.kaniserver.net/doc/cpp/Exceptional%20C%2B%2B.pdf

Sutter, H. 2002: *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*, Addison Wesley, 2002

Wegner, P. 1990: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOPS Messenger,* v.1, i.1, 7-87.

Yaroshenko, O. 1995: *The Beginner's Guide to C++*, Wrox Press Inc., NewYork, NY, US.