project

6

Φοιτητές και Μαθήματα με Κληρονομιές

Περιεχόμενα:

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 αντικειμένου. 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:

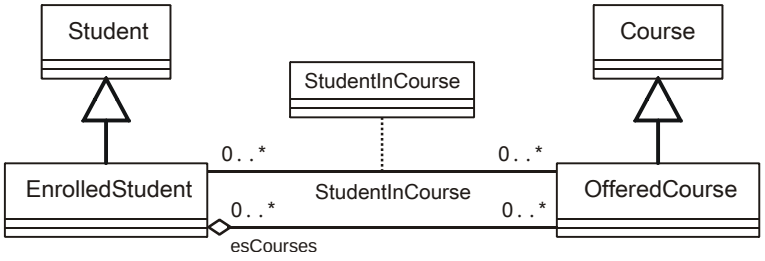
```
      CIndexEntry* cIndex;
      SIndexEntry* sIndex;
      try
      {  cIndex = new CIndexEntry[allCourses.getNOfCourses()];  }
      catch( bad_alloc )
      {  throw ProgXptn( "main", ProgXptn::allocFailed );  }
      try
      {  sIndex = new SIndexEntry[allStudents.getNOfStudents()];  }
      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.