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 );
```

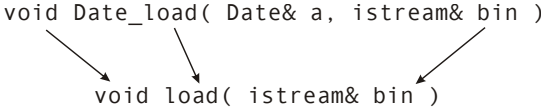
Θα πρέπει λοιπόν να τη δηλώσουμε μέσα στην κλάση:

```
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
```

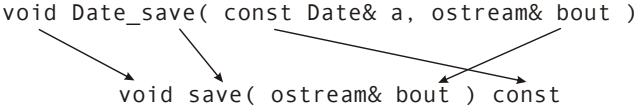
Το νέο στοιχείο εδώ είναι

- 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:



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 $0 < dMonth \leq 12$ και
- ο *dayErr* στην παραβίαση της $0 < dDay \leq lastDay(dYear, dMonth)$.

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", δηλσδή: 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* ⇒ ($bEnergy_{\text{τελική}}$ == $bEnergy_{\text{αρχική}}$ - *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

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

2

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

3

```
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.