κεφάλαιο

21

Ειδικές Συναρτήσεις και Άλλα

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

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

Έννοιες κλειδιά:

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

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

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

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

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

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

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

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

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

  Εδώ θα τα δούμε πληρέστερα.

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

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

21.1 Ερήμην Δημιουργός

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

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

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

δηλαδή, απομονώνουμε την περίπτωση:

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

Οι εντολές:

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

θα μας δώσουν:

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

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

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

Καταλαβαίνεις ότι σπανίως θα γράψεις κλάση χωρίς ερήμην δημιουργό.

Παρατηρήσεις: ▶

1. Γιατί στον δημιουργό με αρχική τιμή γράψαμε:

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

και όχι

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

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

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

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

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

21.1.1 Δημιουργός με Αρχική Τιμή

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

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

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

ορίζουμε:

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

και οι εντολές:

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

θα μα ς δώσουν:

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

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

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

Αποτέλεσμα:

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

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

Βλέπουμε λοιπόν ότι ο δημιουργός με αρχική τιμή καλείται όταν:

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

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

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

και ορίζουμε:

```
BString::BString( const char* rhs )
{
   bsLen = cStrLen( rhs );
   bsReserved = ((bsLen+1)/bsIncr+1)*bsIncr;

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

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

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

και ο δημιουργός της *BString* αν τον δηλώσεις:

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

21.2 Δημιουργός Αντιγραφής

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

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

ή

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

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

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

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

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

Πότε γράφουμε δημιουργό αντιγραφής;

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

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

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

Από τις κλάσεις που είδαμε μέχρι τώρα:

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

Τώρα πρόσεξε το εξής:

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

Στη συνέχεια θα καταλάβεις το γιατί.

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

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

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

Ας πούμε ότι έχουμε μια συνάρτηση:[1]

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

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

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

και στον δημιουργό αντιγραφής:

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

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

```
// . . .
```

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

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

και παίρνουμε:[2]

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

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

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

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

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

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

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

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

Θα πρέπει λοιπόν να θυμάσαι ότι:

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

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

[2] Borland C++, v.5.5

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

21.3 Σειρά Δημιουργίας – Λίστα Εκκίνησης

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

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

  Αυτό φαίνεται και στο παρακάτω

Παράδειγμα 1

Το πρόγραμμα:

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

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

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

θα δώσει:

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

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

Μετά αρχίζει η δημιουργία του *d*:

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

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

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

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

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

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

Παράδειγμα 2

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

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

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

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

Πρόσεξε ότι τώρα:

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

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

Παράδειγμα 3

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

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

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

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

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

και

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

21.4 Εξαιρέσεις από τον Δημιουργό

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

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

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

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

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

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

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

---

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

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

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

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

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

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

και τελικώς θα έχουμε:

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

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

Γενικώς λοιπόν:

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

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

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

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

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

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

Παρατήρησεις: ▶

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

21.5 Ο Καταστροφέας

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

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

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

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

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

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

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

και είναι δεκτός. Γενικώς:

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

Αν έχουμε μια κλάση *K*:

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

Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα

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

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

Με ποιες προδιαγραφές γράφουμε τον καταστροφέα;

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

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

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

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

και δοκιμάζουμε το

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

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

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

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

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

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

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

Αποτέλεσμα:

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

Τι βλέπουμε εδώ;

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

Πώς γίνεται η καταστροφή του αντικειμένου;

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

Με αυτά που ξέρουμε μέχρι τώρα:

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

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

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

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

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

και η εκτέλεση του προγράμματος θα μας δώσει:

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

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

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

21.5.1 Ο Καταστροφέας *δεν* Ρίχνει Εξαιρέσεις

Ένα θέμα που χρειάζεται προσοχή είναι οι εξαιρέσεις του καταστροφέα:

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

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

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

21.5.2 Καλούμε τον Καταστροφέα;

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

Αλλάζοντας λίγο το παράδειγμα που δίναμε εκεί, γράφουμε:

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

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

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

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

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

```
   delete[] buf;
```

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

21.6 Ο Τελεστής Εκχώρησης

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

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

Κάναμε μάλιστα την επισήμανση: «*Εδώ πρόσεξε ότι παίρνουμε τη μνήμη με ένα βοηθητικό βέλος (tmp). Ακόμη, κατά τη συνήθειά μας, βάλαμε εντολές που πιάνουν πιθανή εξαίρεση bad_alloc και ρίχνουν δική μας BStringXptn (allocFailed). Οι εντολές που ακολουθούν δεν θα εκτελεσθούν αν δεν παραχωρηθεί η μνήμη που ζητούμε αφού στην περίπτωση αυτή θα ριχτεί εξαίρεση.*» Συνεπώς, αν κατά την εκτέλεση της "`a = b`" «μείνουμε» από μνήμη θα ριχτεί εξαίρεση αλλά δεν θα καταστραφεί η παλιά τιμή της *a*. Όπως καταλαβαίνεις, έτσι αφήνουμε αρκετές επιλογές –στο πρόγραμμα που χρησιμοποιεί την κλάση– για τη διαχείριση της κατάστασης.

Θα τηρούμε λοιπόν γενικώς τον εξής προγραμματιστικό κανόνα:

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

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

21.6.1 Η Ασφαλής *swap()*

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

---

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

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

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

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

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

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

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

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

Πρόσεξε τα εξής:

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

Τώρα, μπορούμε να γράψουμε:[8]

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

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

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

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

---

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

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

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

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

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

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

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

Παρατηρήσεις: ▶

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

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

Στη συνέχεια θα χρησιμοποιούμε αυτήν τη μορφή.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ή απλώς: `swap( tmp )`.

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

Παρατηρήσεις: ▶

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

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

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

21.7 * Προσωρινά Αντικείμενα

Στην §19.1 είδαμε την εντολή:

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

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

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

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

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

21.8 Ο «Κανόνας των Τριών»

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

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

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

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

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

21.9 Μια Παρένθεση για τη *renew()*

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

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

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

Ας ξαναγράψουμε τα παραπάνω πιο προσεκτικά:

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

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

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

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

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

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

Στην πρώτη περίπτωση θα έχουμε:

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

και στη δεύτερη:

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

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

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

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

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

- Αν δεν μπορέσουμε να πάρουμε δυναμική μνήμη για τον δυναμικό πίνακα που δείχνει η *temp* θα πάρουμε *bad_alloc*.

- Ο ερήμην δημιουργός δημιουργεί τα στοιχεία του δυναμικού πίνακα με τις ερήμην καθορισμένες τιμές. Δεν θα μπορέσει να δημιουργήσει κάποια από τα στοιχεία του δυναμικού πίνακα αν δεν μπορέσει να δεσμεύσει κάποιον πόρο του συστήματος. Στα παραδείγματά μας ο μόνος πόρος που διαχειριζόμαστε είναι η δυναμική μνήμη· άρα θα πάρουμε και πάλι *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.
- Ο αντιγραφικός τελεστής εκχώρησης αντιγράφει το *έγκυρο* αντικείμενο `p[k]` στο `temp[k]`. Η μόνη περίπτωση αποτυχίας της αντιγραφής έχει να κάνει με αποτυχία δέσμευσης πόρων. Για τα παραδείγματά μας αυτό σημαίνει αποτυχία δέσμευσης δυναμικής μνήμης, δηλαδή *bad_alloc* ή *TXptn* με κωδικό *allocFailed*.

  Μπορούμε λοιπόν να χρησιμοποιούμε της εξής μορφή:

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

όπου θεωρούμε ότι το μόνο πράγμα που μπορεί να «πάει στραβά» είναι η αποτυχία παραχώρησης δυναμικής μνήμης.

Αν χρησιμοποιήσεις τη *renew()* όταν τα αντικείμενα της *T* δεσμεύουν άλλους πόρους θα πρέπει να χρησιμοποιήσεις την πιο πολύπλοκη μορφή.[9]

Να τονίσουμε ότι όλες οι παραπάνω μορφές της *renew()* είναι ασφαλείς ως προς τις εξαιρέσεις με την εξής έννοια: αν ριχτεί εξαίρεση

- δεν έχουμε διαρροή μνήμης και
- δεν αλλάζει ο αρχικός πίνακας.

21.10 Αυτά που Μάθαμε στην Πράξη: AN ΔΕ {ΔΑ ΤΕ ΚΑ} GE SE

Στην §20.8 παραθέσαμε μερικά πράγματα που πρέπει να σκεφθούμε όταν γράφουμε μια κλάση. Τώρα μπορούμε να τα δούμε πιο συστηματικά.

Όπως θα πρέπει να έγινε φανερό, από όσα είπαμε μέχρι τώρα, για κάθε κλάση, πέρα από οποιεσδήποτε άλλες ειδικές απαιτήσεις, θα πρέπει να εξετάζουμε –και να υλοποιούμε όταν χρειάζεται– τα εξής:

- αναλλοίωτη (**AN**, class invariant),
- ερήμην δημιουργό (**ΔΕ**, default constructor),

[9] που θα πρέπει να την ολοκληρώσεις.

- δημιουργό αντιγραφής (**ΔΑ**, copy constructor),
- τελεστή εκχώρησης (**ΤΕ**, "**=**"),
- καταστροφέα αντικειμένων (**ΚΑ**, destructor),
- συναρτήσεις "*get*" (**GE**) για όλα τα χαρακτηριστικά,
- συναρτήσεις "*set*" (**SE**) για όλα τα χαρακτηριστικά.

Πάντοτε ή όταν μας χρειάζονται στην εφαρμογή-πελάτη της κλάσης; Είναι πολύ λίγες οι περιπτώσεις που η χρήση μιας κλάσης περιορίζεται σε μια μόνον εφαρμογή. Η εφαρμογή που θα χρησιμοποιήσει για πρώτη φορά μια κλάση μπορεί απλώς να σε οδηγήσει στην υλοποίηση επιπλέον μεθόδων ή/και στην αχρήστευση μερικών μεθόδων "*get*" ή/και "*set*". Αυτό το τελευταίο το είδαμε ήδη στην κλάση *Battery*: δεν γράψαμε μεν *getEnergy*, *setEnergy*, *setMaxEnergy*, *setVoltage* αλλά γράψαμε μεθόδους που –μαζί με τον δημιουργό– δίνουν μεγαλύτερη λειτουργικότητα.

Ας τα πάρουμε ένα-ένα:

Αναλλοίωτη (ΑΝ, class invariant): Η αναλλοίωτη είναι μέρος

- της απαίτησης κάθε δημιουργού (αλλιώς: ισχύει αμέσως μετά τη δημιουργία του κάθε αντικειμένου),
- της προϋπόθεσης του καταστροφέα,
- της προϋπόθεσης και της απαίτησης κάθε μεθόδου ενός αντικειμένου που έχει δηλωθεί "`public`".

Έτσι, μεταξύ άλλων είναι απαραίτητη για να γράψουμε σωστά τις μεθόδους *set* και δημιουργούς με αρχικές τιμές. Θα πρέπει να γράφεται με μαθηματική διατύπωση; Όχι κατ' ανάγκη· μπορεί να καταγραφεί και σε μορφή κειμένου. Αλλά, όσο ακριβέστερα κατάγραφεί τόσο καλύτερα χρησιμοποιείται.

Ερήμην Δημιουργός (ΔΕ, default constructor): Ο ερήμην δημιουργός καλείται για τη δημιουργία: στατικών ή δυναμικών μεταβλητών χωρίς αρχική τιμή, και στατικών ή δυναμικών πινάκων. Σχεδόν απαραίτητος. Αν δεν τον γράψεις θα γράψει έναν εννοούμενο ο μεταγλωττιστής· τα αντικείμενά σου θα ξεκινούν με μη καθορισμένη τιμή που, πιθανότατα, δεν συμμορφώνεται με την αναλλοίωτη. Συνήθως, μπορεί να δηλωθεί μαζί με τον δημιουργό με αρχικές τιμές («2 σε 1»).

Δημιουργός Αντιγραφής (ΔΑ, copy constructor): Καλείται α) όταν κατά τη δήλωση ενός αντικειμένου του δίνουμε ως αρχική τιμή την τιμή ενός άλλου της ίδιας κλάσης, β) για πέρασμα παραμέτρου τιμής σε συνάρτηση γ) επιστροφή τιμής από συνάρτηση με τύπο. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά αρκεί η αυτόματη αντιγραφή; Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Τελεστής Εκχώρησης (ΤΕ, "="): Εκχωρεί την τιμή μιας παράστασης σε μια μεταβλητή (αντικείμενο) του ίδιου τύπου. Αν δεν τον γράψεις θα γράψει έναν συναγόμενο ο μεταγλωττιστής· αλλά δεν είναι σίγουρο ότι κάνει σωστά τη δουλειά. Απαραίτητος ο ορισμός του όταν χρησιμοποιείται δυναμική μνήμη (ή άλλοι πόροι του συστήματος).

Καταστροφέας (ΚΑ, destructor): Ο καταστροφέας είναι απαραίτητος για αντικείμενα που παίρνουν δυναμικά πόρους τη διάρκεια της ζωής τους, π.χ. μνήμη για έναν δυναμικό πίνακα. Δεν χρειάζεται να τον γράψεις όταν η αυτόματη καταστροφή του αντικειμένου κάνει σωστά τη δουλειά.

Συναρτήσεις *get* (GE) για όλα τα χαρακτηριστικά: Εξετάζουμε την ανάγκη για όλα τα χαρακτηριστικά αλλά δεν γράφουμε για όλα οπωσδήποτε! Π.χ. για την *BString* δεν υπάρχει λόγος να γράψουμε μια *getReserved()* που θα επιστρέφει την τιμή της *bsReserved*: αυτή η μεταβλητή είναι ένα «μυστικό» της υλοποίησης που κάνουμε. Να σημειώσουμε ότι, μερικές φορές, αυτές οι μέθοδοι δεν έχουν όνομα *getΚάτι*· παράδειγμα η *BString*::*length()* που είναι στην πραγματικότητα η *getLen* και η *BString*::*c_str()* που είναι η *getData*.

Συναρτήσεις *set* (SE) για όλα τα χαρακτηριστικά: Για όλα; Όχι! Για παράδειγμα, στη *BString* δεν είναι δυνατόν να επιτρέψουμε τη αυθαίρετη μεταβολή των *bsLen* και *bsReserved*. Οι συναρτήσεις *set* μεταβάλλουν τις τιμές μελών και οι νέες τιμές πρέπει να συμμορφώνονται με την αναλλοίωτη. Οι τιμές ορισμένων μελών μπορεί να αλλάζουν από άλλες μεθόδους σύμφωνα με τη λειτουργία του αντικειμένου.

Αν κάθε αντικείμενο έχει **πίνακα(-ες)** –ή συλλογές αντικειμένων άλλου είδους– τότε εξέτασε την ανάγκη για ύπαρξη μεθόδων για:

- **αναζήτηση**,
- **ανάκτηση**,
- **ενημέρωση** (εισαγωγή, διαγραφή, τροποποίηση)

ενός στοιχείου της (κάθε) συλλογής.

Στη *BString* μπορούμε να κάνουμε ανάκτηση και τροποποίηση με την επιφόρτωση του "`[]`". Το ότι αυτό φαίνεται πολύ βολικό και φυσικό για αυτήν την περίπτωση δεν σημαίνει ότι μπορεί να εφαρμοσθεί σε κάθε πίνακα.

Παρατηρήσεις:

1. Μερικοί προγραμματιστές θεωρούν ότι όταν έχουμε να γράψουμε συναρτήσεις *get* και *set* καλύτερα να μη γράψουμε κλάση αλλά δομή.[10] Ο δικός μας σχετικός κανόνας (§19.5) είναι λίγο διαφορετικός και βασίζεται στην αναλλοίωτη.[11] Τον υπενθυμίζουμε: **Αν μια κλάση**
   - **έχει αναλλοίωτη `true` (τα πάντα δεκτά) και**
   - **δεν έχει «μυστικά» της υλοποίησης, δηλαδή μέλη στα οποία δεν θέλουμε να δώσουμε άμεση πρόσβαση,**

   θα προτιμούμε να ομαδοποιούμε τα στοιχεία μας με μια δομή.
2. Πολλοί προγραμματιστές γράφουν καταστροφέα πάντοτε, ακόμη και αν δεν κάνει οτιδήποτε. Π.χ. θα έγραφαν: "`~Date() { }`".[12] Αυτό θα κάνουμε και εμείς από εδώ και πέρα.
3. Φαίνεται ότι ο *Δημιουργός Αντιγραφής*, ο *Τελεστής Εκχώρησης* και ο *Καταστροφέας* πάνε μαζί: ή έχεις να γράψεις και τα τρία (αν παίρνουμε πόρους του συστήματος) ή τίποτε (κανόνας των τριών). Έτσι, έχουν τα πράγματα συνήθως αλλά όχι πάντοτε. Θα δούμε και περιπτώσεις που χρειαζόμαστε μόνον κάποια από αυτά.
4. Φυσικά, η κλάση δεν περιορίζεται στα παραπάνω. Οι πιο ενδιαφέρουσες μέθοδοι θα προκύψουν από τις προδιαγραφές του προβλήματος ή τα σχέδια των εφαρμογών που θα χρησιμοποιήσουν την κλάση.

Στη συνέχεια θα εφαρμόσουμε τους παραπάνω κανόνες και θα ξαναγράψουμε μια κλάση από τα παλιά.

21.10.1 * Επιστροφή στις *string* και *BString*: Μέθοδος *reserve()*

Εδώ –και πριν δώσουμε το παράδειγμά μας– θα πρέπει να κάνουμε μια παρένθεση διότι όποιος «ερευνά τας γραφάς» γίνεται καχύποπτος: «*να μη γράψουμε getReserved() και setReserved(); Και γιατί η string έχει τις μεθόδους capacity() και reserve();*»

Η C++ δίνει –στον προγραμματιστή που χρησιμοποιεί τη *string*– τη δυνατοτητα να ελαχιστοποιήσει τις αλλαγές του δυναμικού πίνακα και –επομένως– τις αντιγραφές που

---

[10] Π.χ. στο (Lockheed Martin 2005) ο *AV Rule 123* λέει: "*The number of accessor and mutator functions **should** be minimized*", δηλαδή: *ο αριθμός των συναρτήσεων get και set πρέπει να ελαχιστοποιείται.*

[11] Να υπενθυμίσουμε ότι και στο (Lockheed Martin 2005) ο *AV Rule 66* λέει: "*A class **should** be used to model an entity that maintains an invariant.*"

[12] Συνηθέστατα: "`virtual ~Date() { }`"· θα το καταλάβεις αργότερα.

γίνονται σε κάθε αλλαγή. Πώς; Με το να καθορίσει το μεγεθος του πίνακα καλώντας τη μέθοδο *reserve()*. Δες ένα παράδειγμα:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s1( "test text" );

   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;

   s1.reserve( 40 );
   cout << s1
        << "  " << s1.length() << "  " << s1.capacity() << endl;
}
```

Αποτέλεσμα:

```
test text  9  9
test text  9  40
```

Η "`s1.capacity()`" μας επιστρέφει ως τιμή (`size_t`)[13] το μέγιστο μήκος που μπορεί να φτάσει το κείμενο που είναι αποθηκευμένο στην *s1* χωρίς να χρειαστεί να αλλάξει ο δυναμικός πίνακας. Με την "`s1.reserve(40)`" αλλάζουμε σε "`40`" (αυτό) το μέγιστο μήκος.

Η "`s1.reserve(` *n* `)`"

- Δεν θα αλλάξει την τιμή της *s1*.
- Αν *n* > *s1*.*capacity()* θα αλλάξει το μέγιστο μήκος σε *n* (ή μεγαλύτερο) εκτός αν δεν βρει αρκετή μνήμη ή *n* > *s1*.*max_size()*.
- Αν *n* < *s1*.*capacity()* δεν είναι απαραίτητο να αλλάξει κάτι.

Ας γράψουμε τις δύο αυτές μεθόδους για τη *BString*. Να παραδεχτούμε ότι η *capacity()* είναι η *getReserved()*:

```
   size_t capacity() const {  return bsReserved;  }
```

Η *reserve()* είναι πιο περίπλοκη:

```
void BString::reserve( int newRes )
{
   if ( newRes < 0 )
      throw BStringXptn( "reserve",
                         BStringXptn::domainError, newRes );
   if ( newRes > bsReserved ||
        (newRes < bsReserved/2 && newRes > bsLen) )
   {
      char* tmp;
      newRes = (newRes/bsIncr+1)*bsIncr;
      try {  tmp = new char[newRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "reserve", BStringXptn::allocFailed ); }
      bsReserved = newRes;
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
} // BString::reserve
```

Πρόσεξε ότι αλλάζουμε την τιμή του *bsReserved* αν:

- *newRes* > *bsReserved* ή
- *bsLen* < *newRes* < *bsReserved*/2

Ακόμη, για να διατηρήσουμε την αναλλοίωτή μας αλλάζουμε την τιμή της *newRes* σε "`(newRes/bsIncr+1)*bsIncr`".

---

[13] Ακριβέστερα: `string::size_type`.

Τέλος, πρόσεξε ότι αν η *reserve()* κληθεί με αρνητικό όρισμα ρίχνουμε εξαίρεση με κωδικό λάθους *BStringXptn*::*domainError* ενώ η *string*::*reserve()* ρίχνει (και) σε αυτήν την περίπτωση *bad_alloc*.[14]

Όπως βλέπεις, αυτήν τη μέθοδο δεν τη λές απλώς «*setReserved*»!

Αν προσαρμόσεις το παραπάνω προγραμματάκι στη *BString* θα πάρεις:

```
test text  9  16
test text  9  48
```

21.11 Λίστα με Απλή Σύνδεση

Στο Παράδ. 3 της §16.13 είδαμε μια *λίστα με απλή σύνδεση* (τύπος *SList*), με περιεχόμενο αντικείμενα τύπου *GrElmn*. Τώρα, θα ξαναλύσουμε το πρόβλημα γράφοντας την κλάση *SList* με εφαρμογή της συνταγής που είδαμε στην προηγούμενη παράγραφο.

Αναλλοίωτη (ΑΝ): Η λίστα είναι ένα σύνολο κόμβων που για τον καθένα –ας πούμε *aNode*– ισχύει το εξής:

είναι πρώτος κόμβος της λίστας (`*`*slHead* `==` *aNode*)[15]

ή (αποκλειστικώς, **xor**)

υπάρχει ένας και μόνον ένας άλλος κόμβος *bNode* που είναι προηγούμενος του *aNode*

(`*(`*bNode*`.`*lnNext*`)` `==` *aNode*)[16]

Ακόμη:

υπάρχει ένας και μόνον ένας κόμβος που δεν έχει επόμενο και

τον δείχνει το *slTail* (*slTail*`->`*lnNext* `==` 0)

Αυτά ισχύουν για κάθε λίστα με απλή σύνδεση.

Για τη συγκεκριμένη λίστα:

- Το περιεχόμενο της είναι ένα *σύνολο* –ας το πούμε *content*– αντικειμένων τύπου *GrElmn*:
  - Σε κάθε κόμβο της λίστας, εκτός από τον `*`*slTail*, υπάρχει ένα και μόνον ένα στοιχείο του *content*. Αφού το *content* είναι σύνολο θα έχουμε:

    (*aNode* `!=` *bNode*) ⇒ (*aNode*`.`*lnData* `!=` *bNode*`.`*lnData*)
- Η σχέση *content* – κόμβων είναι ολική συνάρτηση (αλλά όχι "*επί*", αφού υπάρχει ο φρουρός στο `*`*slTail*).

Επομένως, η κλάση μας έχει μη τετριμμένη αναλλοίωτη.

Από τα παραπάνω γίνεται σαφές και κάτι άλλο: ο τύπος *ListNode* έχει θέση *μέσα* στην *SList*.

Έτσι, θα υλοποιήσουμε την κλάση μας ως:

```
class SList
{
public:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
}; // SList
```

---

[14] Δεν εξετάζουμε την περίπτωση: *n* > *max_size()*.

[15] Ή: *slHead* `==` `&`*aNode*.

[16] Ή: *bNode*`.`*lnNext* `==` `&`*aNode*.

Όπως καταλαβαίνεις, έξω από την κλάση ο τύπος του κόμβου θα πρέπει να γράφεται "`SList::ListNode`".

Ερήμην Δημιουργός (ΔΕ): Πρέπει να γράψουμε Ερήμην Δημιουργό για να βάλουμε τον φρουρό. Θα κρατήσουμε αυτόν που γράψαμε στην §16.13 με μια αλλαγή: Δεν θα ρίχνουμε εξαίρεση *ApplicXptn* αλλά *SListXptn*:

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

Δημιουργός Αντιγραφής (ΔΑ): Κάθε λίστα δεσμεύει πόρους –συγκεκριμένα: δυναμική μνήμη– του συστήματος. Επομένως χρειάζεται να γράψουμε εμείς τον (σωστό) δημιουργό αντιγραφής:

```
SList::SList( const SList& other )
{
   try
   {
      slHead = new ListNode;
      slTail = slHead;
      ListNode* p( other.slHead );
      while ( p != other.slTail )
      {
         slTail->lnData = p->lnData;
         slTail->lnNext = new ListNode;
         slTail = slTail->lnNext;
         p = p->lnNext;
      }
      slTail->lnNext = 0;
   }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
} // SList::SList
```

Και είναι σωστός αυτός ο δημιουργός αντιγραφής; Ας πούμε ότι έχουμε να αντιγράψουμε μια λίστα με 200 στοιχεία και παίρνουμε *bad_alloc* όταν προσπαθήσουμε να πάρουμε μνήμη για το 110το στοιχείο· με τον παραπάνω δημιουργό θα έχουμε διαρροή μνήμης που πήραμε για τα πρώτα 109 στοιχεία! Ας την ξαναγράψουμε πιο προσεκτικά:

```
SList::SList( const SList& other )
{
   try {  slHead = new ListNode;  }
   catch( bad_alloc& )
   {  throw SListXptn( "SList", SListXptn::allocFailed );  }
   slTail = slHead;
   ListNode* p( other.slHead );
   while ( p != other.slTail )
   {
      slTail->lnData = p->lnData;
      try {  slTail->lnNext = new ListNode;  }
      catch( bad_alloc& )
      {
         while ( slHead != slTail )
         {
            ListNode* p( slHead );
            slHead = slHead->lnNext;
            delete p;
         }
         delete slHead;
         throw SListXptn( "SList", SListXptn::allocFailed );
      }
      slTail = slTail->lnNext;
      p = p->lnNext;
```

```
   }
   slTail->lnNext = 0;
} // SList::SList
```

Όπως βλέπεις, στην "`catch`" το πρώτο που κάνουμε είναι να επιστρέψουμε τη μνήμη που πήραμε για τους κόμβους που ήδη αντιγράψαμε· μετά ρίχνουμε την εξαίρεση. Αυτός ο δημιουργός είναι σωστός και έχει *βασική εγγύηση ασφάλειας* ως προς τις εξαιρέσεις.

Καταστροφέας (ΚΑ): Αφού το κάθε αντικείμενο έχει δεσμεύσει δυναμική μνήμη πρέπει να γράψουμε καταστροφέα που θα την ανακυκλώνει:

```
SList::~SList()
{
   while ( slHead != slTail )
   {
      ListNode* p( slHead );
      slHead = slHead->lnNext;
      delete p;
   }
   delete slHead;
   slTail = slHead = 0;
} // SList::~SList
```

Εδώ βλέπεις έναν καταστροφέα που δεν έχει «μόνο μια γραμμή». Φυσικά, δεν είναι τίποτε άλλο από την *SList_deleteAll()*.

Τελεστής Εκχώρησης (ΤΕ): Αφού γράψαμε δημιουργό αντιγραφής και καταστροφέα θα πρέπει να γράψουμε και τελεστή εκχώρησης:

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

όπου:

```
void SList::swap( SList& other )
{
   std::swap( slHead, other.slHead );
   std::swap( slTail, other.slTail );
} // SList::swap
```

Συναρτήσεις *get* (GE): «Καλώς εχόντων των πραγμάτων» δεν θα έπρεπε να δώσουμε συναρτήσεις "*get*". Πάντως, η *list* της STL δίνει παρόμοιες μεθόδους και αντί για *getHead()* και *getTail()* τις ονομάζει *begin()* και *end()* αντιστοίχως. Αυτό θα κάνουμε και εδώ. Τις ορίζουμε `inline`:

```
   ListNode* begin()  {  return slHead;  }
   ListNode* end()  {  return slTail;  }
```

Στο επόμενο κεφάλαιο θα δούμε και μια περίπτωση που μπορούμε να τις χρησιμοποιήσουμε αφού τις αλλάξουμε λίγο.

Συναρτήσεις *set* (SE): Φυσικά αποκλείεται να δώσουμε δικαίωμα για αλλαγή τιμών των δύο μελών αφού κάτι τέτοιο θα ήταν καταστροφικό για τη λίστα.

Διαχείριση Περιεχομένου: Θα γράψουμε μεθόδους διαχείρισης του περιεχομένου όπως περίπου δουλέψαμε και στη *Route*. Η διαχείριση του περιεχομένου θα γίνεται με βάση τον ατομικό αριθμό που τον επιλέξαμε ως κλειδί για τον *GrElmn*. Θα ξεκινήσουμε με την αντίστοιχη της *findNdx()*. Η συνάρτηση που θα γράψουμε θα επιστρέφει βέλος και όχι δείκτη αφού τώρα δεν έχουμε πίνακα:

```
SList::ListNode* SList::findPtr( int aAn ) const
{
   ListNode* fv;
   if ( aAn <= 0 )
```

```
      fv = slTail;
   else
   {
      slTail->lnData = GrElmn( aAn );
      fv = slHead;
      while ( (fv->lnData).getANumber() != aAn )  fv = fv->lnNext;
   }
   return fv;
} // SList::findPtr
```

Πρόσεξε τα εξής:

1. Η μέθοδος ψάχνει να βρει κόμβο που να έχει στο *lnData* τα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*. Αν βρει τέτοιο κόμβο επιστρέφει βέλος προς αυτόν· αλλιώς επιστρέφει βέλος προς τον φρουρό (== *slTail*).
2. Αν πάρει μη θετικό ακέραιο ως όρισμα δεν ρίχνειε εξαίρεση αλλά αναφέρει «δεν το βρήκα.»
3. Αυτή η μέθοδος είναι η αρχή της εγκατάλειψης της λειτουργίας της στοίβας που είχαμε στην αρχική λίστα. Σε μια «γνήσια» στοίβα δεν υπάρχει διαδικασία αναζήτησης· το μόνο που βλέπουμε είναι η κορυφή.

Αναζήτηση: Η *findPtr()*, όπως και η *findNdx()*, θα είναι **private**. Η αναζήτηση από ένα πρόγραμμα-πελάτη θα γίνεται με τη μέθοδο:

```
bool SList::find1Elmn( int aAn ) const
{
   return ( findPtr(aAn) != slTail );
} // SList::find1Elmn
```

που επιστρέφει "`true`" αν βρει στη λίστα δεδομένα για το στοιχείο με ατομικό αριθμό *aAn*.

Ανάκτηση: Και η ανάκτηση των δεδομένων για ένα στοιχείο γίνεται με βάση τον ατομικό αριθμό:

```
const GrElmn& SList::get1Elmn( int aAn ) const
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode == slTail )
      throw SListXptn( "get1Elmn", SListXptn::notFound, aAn );
   return ptrToNode->lnData;
} // SList::get1Elmn
```

Όπως κάναμε στην *get1RouteStop()* έτσι και εδώ, αν δεν βρούμε το στοιχείο που ζητείται, ρίχνουμε εξαίρεση. Αυτό σημαίνει ότι θα πρέπει να έχουμε εξασφαλίσει την ύπαρξη των δεδομένων πριν δώσουμε το μήνυμα *get1RouteStop()*.

Εισαγωγή: Η *push_front()* δεν μας εξασφαλίζει τη μοναδικότητα των αντικειμένων που περιέχει η λίστα. Γράφουμε λοιπόν την:

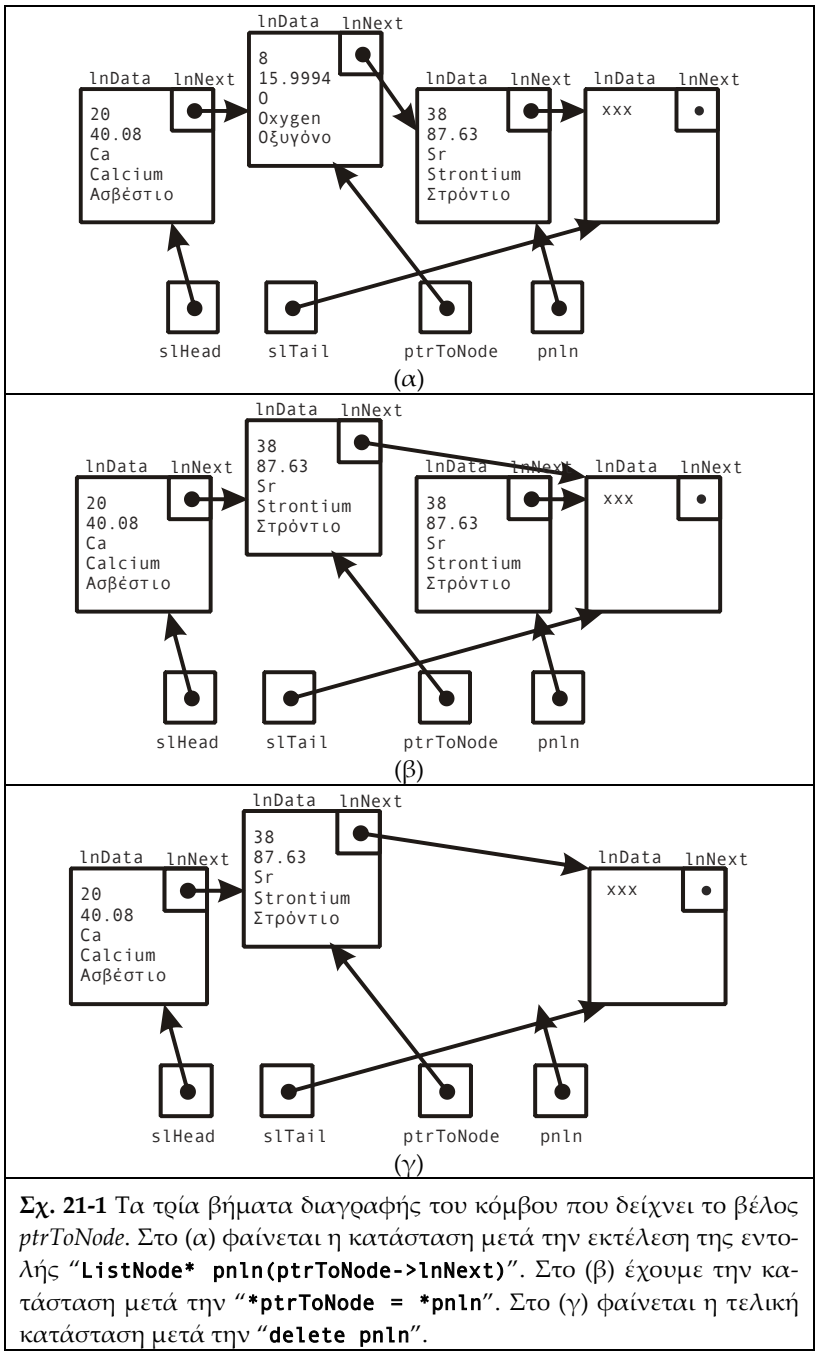
```
void SList::insert1Elmn( const GrElmn& aItem )
{
   ListNode* ptrToNode( findPtr(aItem.getANumber()) );
   if ( ptrToNode == slTail ) // δεν υπάρχει
   {
      push_front( aItem );
   }
} // SList::insert1Elmn
```

Αυτή καλεί την *push_front()* αφού έχει εξασφαλίσει ότι τα δεδομένα που θέλουμε να εισαγάγουμε δεν υπάρχουν ήδη στη λίστα. Για να αποτρέψουμε κακή χρήση της *push_front()* που θα παραβίαζε την αναλλοίωτη θα πρέπει να την «κρύψουμε» σε περιοχή **private**.

Όπως στην *insert1RouteStop()*, αν βρούμε το προς εισαγωγή στοιχείο στη λίστα δεν κάνουμε οτιδήποτε άλλο.

Διαγραφή: Εδώ έχουμε μεγάλη απόκλιση από τη λογική στοίβας: στη στοίβα η μόνη διαγραφή που μπορεί να γίνει είναι αυτή της κορυφής (*pop_front*). Εδώ, αν μας ζητηθεί να διαγράψουμε τα δεδομένα του στοιχείου με ατομικό αριθμό *aAn*, το αναζητούμε:

```
   ListNode* ptrToNode( findPtr(aAn) );
```



Σχ. 21-1 Τα τρία βήματα διαγραφής του κόμβου που δείχνει το βέλος *ptrToNode*. Στο (α) φαίνεται η κατάσταση μετά την εκτέλεση της εντολής "`ListNode* pnln(ptrToNode->lnNext)`". Στο (β) έχουμε την κατάσταση μετά την "`*ptrToNode = *pnln`". Στο (γ) φαίνεται η τελική κατάσταση μετά την "`delete pnln`".

και αν το βρούμε (*ptrToNode* `!=` *slTail*) πρέπει να διαγράψουμε τον κόμβο που δείχνει το *ptrToNode*.

Υπάρχει όμως ένα πρόβλημα: θα πρέπει να αλλάξουμε την τιμή του *lnNext* του *προηγούμενου* κόμβου που δεν ξέρουμε ποιος είναι! Θα κάνουμε κάτι άλλο:

```
ListNode* pnln( ptrToNode->lnNext );
*ptrToNode = *pnln;
delete pnln;
```

Δηλαδή: *αντιγράφουμε τον επόμενο κόμβο* (υπάρχει πάντοτε, να είναι καλά ο φρουρός) *στον στοχευόμενο και διαγράφουμε τον επόμενο*. Δες το Σχ. 21-1 όπου υποτίθεται ότι θέλουμε (α) να διαγράψουμε το "Οξυγόνο". Αντιγράφουμε (β) τα δεδομένα για το "Στρόντιο" σβήνοντας αυτά του Οξυγόνου. Αντιγράφουμε και το βέλος *lnNext* που τώρα πια δείχνει τον φρουρό. Τελικώς (γ) ανακυκλώνουμε τον κόμβο που ήταν το "Στρόντιο".

Αν ο κόμβος που ανακυκλώνουμε (*pnln*) είναι ο φρουρός θα πρέπει να αλλάξουμε την τιμή του *slTail*. Στην περίπτωση αυτή δεν υπάρχει λόγος να αντιγράψουμε ολόκληρον τον κόμβο· αρκεί ο μηδενισμός του *lnNext*:

```
void SList::erase1Elmn( int aAn )
{
   ListNode* ptrToNode( findPtr(aAn) );
   if ( ptrToNode != slTail ) // υπάρχει
   {
      ListNode* pnln( ptrToNode->lnNext );
      if ( pnln != slTail )
         *ptrToNode = *pnln;
      else
      {
         ptrToNode->lnNext = 0;
         slTail = ptrToNode;
      }
      delete pnln;
   }
} // SList::erase1Elmn
```

Τροποποίηση: Ο πιο απλός τρόπος να αποφύγουμε παραβίαση της αναλλοίωτης από τροποποίηση δεδομένων ενός στοιχείου είναι αυτός που χρησιμοποιήσαμε και στη *RouteStop*:

```
GrElmn tmp( lst.get1Elmn(n) );
Τροποποίησε το tmp
lst.erase1Elmn( n );
lst.insert1Elmn( tmp );
```

21.11.1 Άλλες Μέθοδοι;

Αυτά που είδαμε μέχρι εδώ ήταν αυτά που βγαίνουν από τη συνταγή μας. Υπάρχουν άλλες ανάγκες; Ναι, η φύλαξη των δεδομένων που έχουμε αποθηκευμένα στη λίστα. Στο αρχικό πρόγραμμα, για τη φύλαξη, καλούσαμε τη συνάρτηση *saveUpdateList()*. Μπορούμε να την τροποποιήσουμε και να τη χρησιμοποιήσουμε και τώρα. Η τροποποίηση έχει να κάνει με το ότι έχουμε «κρύψει» τα δυο μέλη της κλάσης:

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

Ο τύπος της *p* (μεταβλητή-βέλος) με την οποία διασχίζουμε τη λίστα είναι "`const SList::ListNode*`". Αντί για *lst.slHead*, *lst.slTail* έχουμε `lst.begin()`, `lst.end()` αντιστοίχως.

«Καλά», θα πεις, «εδώ προσπαθούμε να κρύψουμε τις λειτουργίες της λίστας και τώρα θα πρέπει να τη διασχίσουμε σε εξωτερική συνάρτηση;! Δεν γίνεται να κάνουμε τη φύλαξη με μια μέθοδο;» Σωστή παρατήρηση! Αλλά σκέψου το εξής: Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. Εδώ, για παράδειγμα, τα αντικείμενα που αποτελούν το περιεχόμενο της λίστας φυλάγονται σε αρχείο τυχαίας πρόσβασης και έχουν ένα μέλος που καθορίζει τη θέση τους μέσα στο αρχείο. Καταλαβαίνεις λοιπόν ότι η διάσχιση της λίστας δεν μπορεί να «κρυφτεί» σε κάποια μέθοδο. Στο επόμενο κεφάλαιο θα δούμε κάποια εργαλεία για να κάνουμε τη διάσχιση χωρίς να βλέπουμε τη «λεπτομέρεια» "`p = p->lnNext`".

Παρ' όλα αυτά, περισσότερο για λόγους επίδειξης, ας κάνουμε μια προσπάθεια να γράψουμε μια μέθοδο που να κάνει αυτή τη δουλειά:

```
void SList::save( ostream& bout,
```

```
                  void (*wrProc)(const GrElmn&, ostream&) ) const
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      (*wrProc)( p->lnData, bout );
} // save
```

Εδώ έχουμε μια μέθοδο που διασχίζει τη λίστα και για κάθε κόμβο καλεί μια συνάρτηση που περνούμε ως παράμετρο. Η κλήση της γίνεται ως εξής:

```
      lst.save( bInOut, writeRandom );
```

Είναι καλύτερη αυτή η λύση; Όπως νομίζεις... Πλεονεκτεί στο ότι έκρυψε τη διάσχιση της λίστας, μειονεκτεί στο ότι περνούμε μια εξωτερική συνάρτηση ως παράμετρο σε μια μέθοδο της κλάσης.

Στη συνέχεια θα προτιμήσουμε τη *saveUpdateList()*.

21.11.2 Το Πρόγραμμα

Το πρόγραμμα τώρα θα είναι ως εξής:

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

#include "GrElmn.cpp"
#include "SList.cpp"

using namespace std;

struct ApplicXptn
// ΟΠΩΣ ΣΤΟ ΑΡΧΙΚΟ

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

void saveUpdateList( fstream& bInOut, const SList& lst );
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

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

   try
   {
      SList   lst;

      openFile( flNm, bInOut );
   // Υπολόγισε τον μέγιστο α.α. (πλήθος εγγραφών αρχείου)
      unsigned int maxAtNo;
      countRecords( bInOut, maxAtNo );
      int aa;
      do {
      // Διάβασε τον α.α.
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            GrElmn tmp;
            if ( lst.find1Elmn(aa) )
            {
               tmp = lst.get1Elmn( aa );
```

```
               lst.erase1Elmn( aa );
            }
            else
               readRandom( tmp, bInOut, aa );
            editGrNameMM( tmp );
            lst.insert1Elmn( tmp );
         }
      } while ( aa != 0 );
      saveUpdateList( bInOut, lst );
      bInOut.close();
      if ( bInOut.fail() )
         throw ApplicXptn( "main", ApplicXptn::cannotClose,
                           flNm.c_str() );
   }
   catch( GrElmnXptn& x )
   // . . .
   catch( SListXptn& x )
   // . . .
   catch( ApplicXptn& x )
   // . . .
} // main
```

Πρόσεξε ότι με την εγκατάλειψη της *elmntInList* το κομμάτι

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

αντικαταστάθηκε από το

```
GrElmn tmp;
if ( lst.find1Elmn(aa) )
{
   tmp = lst.get1Elmn( aa );
   lst.erase1Elmn( aa );
}
else
   readRandom( tmp, bInOut, aa );
editGrNameMM( tmp );
lst.insert1Elmn( tmp );
```

21.12 * Βέλος προς Μέθοδο

Επιστρέφουμε στη μέθοδο *SList*::*save* με την εξής ερώτηση: Η *GrElmn* έχει τρεις μεθόδους – τις *save()*, *display()* και *writeToTable()*– με τις οποίες φυλάγουμε την τιμή του αντικειμένου σε κάποιο αρχείο. Είναι δυνατόν, με χρήση της *SList*::*save()*, να φυλάγουμε το περιεχόμενο της λίστας με οποιονδήποτε από τους τρεις τρόπους; Ναι, αλλά με την προϋπόθεση ότι θα γράψουμε την κατάλληλη εξωτερική συνάρτηση-κέλυφος.

Ας πάρουμε τη *GrElmn*::*display()*. Αυτή έχει μια παράμετρο τύπου *ostream*&. Όταν καλούμε την *SList*::*save* πρέπει να βάλουμε ως δεύτερο όρισμα μια συνάρτηση **void** με δύο παραμέτρους: η πρώτη είναι τύπου "`const GrElmn&`" και η δεύτερη τύπου *ostream*&. Η *GrElmn*::*display()* δεν ταιριάζει! Γράφουμε λοιπόν μια εξωτερική συνάρτηση:

```
void exDisplay( const GrElmn& aElmn, ostream& outF )
{
   aElmn.display( outF );
} // exDisplay
```

Αυτή κάνει τη δουλειά που θέλουμε και ταιριάζει ως δεύτερο όρισμα όταν καλούμε την *SList*::*save()*.

Αυτή φυσικά δεν είναι μια καλή λύση: Για να έχουμε αυτό που θέλουμε θα πρέπει να γράψουμε τρεις εξωτερικές συναρτήσεις, «ενδιάμεσες» μεταξύ *SList* και *GrElmn*.

Αυτό που θέλουμε είναι να γράψουμε μια:

```
void SList::toFile( ostream& out, OutFunc outProc ) const
```

```
{
   for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
      ((p->lnData).*outProc)( out );
} // toFile
```

που όταν καλείται το *outProc* θα είναι βέλος προς τη *save* ή τη *display* ή τη *writeToTable*. Θέλουμε δηλαδή να χρησιμοποιήσουμε της τεχνική *συναρτήσεων ανάκλησης* (callback, §14.4) για μεθόδους. Πώς θα ορίσουμε τον τύπο *OutFunc*; Οι δηλώσεις των τριών μεθόδων στην *GrElmn* είναι:

```
class GrElmn
{
// . . .
   void save( ostream& bout ) const;
   void display( ostream& tout ) const;
   void writeToTable( ostream& tout ) const;
// . . .
}; // GrElmn
```

Να πούμε:

```
typedef void (*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc* δείχνει μια συνάρτηση **void** και **const** με μια παράμετρο τύπου *ostream*&; Όχι! Ο ορισμός πρέπει να είναι λίγο διαφορετικός:

```
typedef void (GrElmn::*OutFunc)( ostream& ) const;
```

δηλαδή: Ένα βέλος τύπου *OutFunc δείχνει μια μέθοδο της GrElmn* **void** και **const** με μια παράμετρο τύπου *ostream*&.

Βάζουμε αυτόν τον ορισμό μετά τον ορισμό της *GrElmn* αλλά πριν τον ορισμό της *SList* και οι εντολές:

```
      ofstream tout( "test.txt" );
      lst.toFile( tout, &GrElmn::display );
      tout.close();
```

θα γράψουν το περιεχόμενο της λίστας στο αρχείο test.txt.

21.13 Μετατροπές Τύπου

Στα παραδείγματά μας μέχρι τώρα είδαμε μερικές περιπτώσεις μετατροπών τύπου:

- Ο δημιουργός `BString(const char* rhs)` παίρνει ένα κείμενο σε μορφή πίνακα χαρακτήρων και μας δίνει έναν αντικείμενο κλάσης *BString* με τιμή το ίδιο κείμενο.
- Αντιστρόφως, η *c_str()* μας δίνει ένα βέλος (`const char*`) προς πίνακα χαρακτήρων (με `'\0'` στο τέλος.)
- Είδαμε τον δημιουργό της *Date* να παίρνει τον φυσικό 2008 και να τον μετατρέπει σε αντικείμενο (01.01.2008)

Επειδή τέτοιες μετατροπές είναι, γενικώς, χρήσιμες οι προγραμματιστές της C++ έχουν παγιώσει την εξής συνταγή:

- Οι μετατροπές τύπου προς μια κλάση γίνονται από δημιουργούς της κλάσης.
- Οι μετατροπές τύπου από μια κλάση προς άλλους τύπους γίνεται με ειδικές συναρτήσεις μέλη της κλάσης.

Φυσικά, κανείς δεν σε εμποδίζει να παραβείς αυτούς τους κανόνες αρκεί να το κάνεις σωστά.

21.13.1 Μετατροπή με Δημιουργό

Λέγαμε πιο πριν (§21.1.1): «*Υπάρχει περίπτωση να έχουμε δημιουργό που χρειάζεται οπωσδήποτε κάποια αρχική τιμή στη δήλωση; Δεν είναι τόσο συχνή περίπτωση αλλά υπάρχει*

πιθανότητα να χρειαστείς κάτι τέτοιο.» Ο **δημιουργός μετατροπής** (conversion constructor) είναι ακριβώς μια τέτοια περίπτωση: ένας δημιουργός με μία παράμετρο που μετατρέπει μια τιμή του τύπου της παραμέτρου σε αντικείμενο της κλάσης που ανήκει. Για την ακρίβεια: ένας δημιουργός μετατροπής *θα πρέπει να μπορεί να κληθεί με ένα όρισμα μόνον*. Αυτό σημαίνει ότι μπορεί να έχει και άλλες παραμέτρους που όμως έχουν ερήμην τιμές.

Παραδείγματα

Μετά τους διαχωρισμούς που κάναμε στην §21.1.1 έχουμε τον

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

που καλείται με ένα και μόνον ένα όρισμα και μετατρέπει μια τιμή "`const char*`"[17] σε αντικείμενο *BString* και έτσι –αν η *a* είναι τύπου *BString*– μπορούμε να δώσουμε:

```
a = BString( "abc" );
```

Έχουμε ακόμη τον

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

που μπορεί να κληθεί με ένα όρισμα και να μετατρέψει μια τιμή "`int`" σε αντικείμενο *Date* και έτσι –αν η *d* είναι τύπου *Date*– με την εντολή:

```
d = Date( 375 );
```

δίνουμε στην *d* τιμή "01.01.375".

Στην §15.14.1 είδαμε τη συνάρτηση

```
GrElmn GrElmn_copyFromElmn( const Elmn& a )
```

–που από μια τιμή *Elmn* μας δίνει μια τιμή *GrElmn*– και υποσχεθήκαμε ότι «*αργότερα θα μάθουμε πώς μπορούμε να δώσουμε καλύτερη λύση.*» Τη δώσαμε στην §19.6 με τον δημιουργό μετατροπής

```
GrElmn::GrElmn( const Elmn& rhs )
```

Παρατήρηση: ▶

Πρόσεξε ότι και οι «2 σε 1» δημιουργοί των *Date* και *BString* είναι δημιουργοί μετατροπής αφού είναι δυνατόν να κληθούν με ένα όρισμα.◀

Τώρα πρόσεξε το εξής: παρ' όλο που δεν έχουμε ορίσει

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

η εκχώρηση

```
a = "abc";
```

εκτελείται χωρίς κανένα πρόβλημα. Γιατί; Διότι καλείται αυτομάτως ο δημιουργός μετατροπής και μετατρέπει το `"abc"` σε αντικείμενο κλάσης *BString*. Δηλαδή και αυτή η εντολή εκτελείται ως "`a = BString("abc")`". Αυτό είναι καλό και βολικό.

Για τη *Date* όμως τα πράγματα αλλάζουν. Αν γράψεις

```
if ( d < 2012 )
{ . . .
```

γίνεται αυτομάτως η σύγκριση "`d < Date(2012)`" που υπολογίζεται ως «η *d* είναι *πριν από* την 01.01.2012» και είναι ακριβώς αυτό που θέλουμε. Αν όμως γράψουμε "`d > 2012`" αυτό θα υπολογιστεί ως «η *d* είναι *μετά* την 01.01.2012» ενώ εμείς θέλουμε να πούμε «η *d* είναι *μετά* την 31.12.2012». Ένα τέτοιο λάθος στο πρόγραμμα είναι πολύ δύσκολο να εντοπιστεί.

---

[17] Μπορούμε να τον κάνουμε να μετατρέπει ένα τμήμα του ορίσματος. Με τον:

```
BString( const char* rhs, int n=0 );
```

δημιουργούμε αντικείμενο που έχει αρχική τιμή το κομμάτι του *rhs* από τη θέση *n* και μετά. Με την "`a = BString("abcde", 2)`" η *a* θα πάρει ως τιμή `"cde"`.

Η C++ μας δίνει τη δυνατότητα να αποφύγουμε τέτοια λάθη. Αλλάζουμε τη δήλωση του δημιουργού μετατροπής:

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

Αυτό το "**explicit**" έχει το εξής νόημα: «*μην κάνεις αυτόματες μετατροπές. Θα κάνεις μετατροπή αν ζητείται ρητώς.*» Δηλαδή δεν είναι δεκτές οι "`d = 375`", "`d < 2012`", "`d > 2012`" και θα πρέπει να γραφούν ως "`d = Date( 375 )`", "`d < Date(2012)`", "`d > Date(2012)`". Έτσι, (υποτίθεται ότι) ο προγραμματιστής θα σκεφτεί καλύτερα αυτό που γράφει.

Παρομοίως, στη *GrElmn* καλό θα ήταν να γράψουμε:

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

ώστε να μη γίνεται ερήμην μας η μετατροπή του "`88`" σε αντικείμενο *GrElmn* που αναφέρεται στο ράδιο.

Και για τη *BString* θα μπορούσαμε να δηλώσουμε:

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

αλλά, όπως είπαμε, στην περίπτωση αυτή η αυτόματη μεταροπή μας βολεύει. Πάντως, γενικώς, καλό θα είναι να ακολουθείς τον *Κανόνα OBJ32* του (CERT 2009):[18]

- ***Εξασφάλισε ότι οι δημιουργοί που μπορεί να κληθούν με ένα όρισμα δηλώνονται*** `explicit`.

21.13.2 Συναρτήσεις Μετατροπής

Οι συναρτήσεις μετατροπής είναι συναρτήσεις-μέλη που μετατρέπουν την τιμή ενός αντικειμένου σε έναν άλλον τύπο. Όπως είπαμε και πιο πάνω, τέτοια δουλειά κάνει και η *c_str()*· η C++ όμως μας δίνει μια πάγια μορφή για τέτοιες συναρτήσεις.

Μια συνάρτηση που κάνει την ίδια δουλειά με τη *c_str()* ορίζεται ως εξής:

```
public:
// . . .
   operator const char*() const
   {  bsData[bsLen] = '\0';  return bsData;  };
```

Ας πούμε ότι έχουμε δηλώσει:

```
      BString q( "abc" );
      char z[30];
```

Όπως ξέρουμε, μπορούμε να γράψουμε:

```
      strcpy( z, q.c_str() );
      cout << z << endl;
```

Τώρα, με τη νέα συνάρτηση, μπορούμε να γράψουμε και:

```
      strcpy( z, q );
      cout << z << endl;
```

Πρόσεξε ότι στη *strcpy* χρησιμοποιούμε το όνομα του αντικειμένου μόνο.

Πάντως, κανείς δεν μας εμποδίζει να γράψουμε:

```
public:
// . . .
   operator const char*()
   {  bsData[bsLen] = '\0';  return bsData;  };
   operator size_t() {  return bsReserved;  };
```

και να το χρησιμοποιήσουμε:

```
      size_t n( q );
```

Βέβαια, δημοκρατία έχουμε και κάνεις ό,τι θέλεις, αλλά μια τέτοια χρήση της δυνατοτητας που μας δίνεται είναι ανόητη.

---

[18] OBJ32: *Ensure that single-argument constructors are marked* "`explicit`".

Τελικώς όμως: Πόσο καλό είναι το ότι μπορούμε και γράφουμε `strcpy(z, q)` αντί για `strcpy(z, q.c_str())`; Ε, δεν είναι και καμιά μεγάλη κατάκτηση. Μπορούμε να ζήσουμε και χωρίς αυτήν τη συνάρτηση· ή *c_str()* είναι σαφώς προτιμότερη.[19]

21.14 Στατικά Μέλη Κλάσης

Ας πούμε τώρα ότι έχουμε το εξής πρόβλημα: θέλουμε να έχουμε τη δυνατότητα να ξέρουμε πόσα αντικείμενα κλάσης *Date* υπάρχουν στο πρόγραμμά μας. Τι κάνουμε; Μια ιδέα είναι να δηλώσουμε μια καθολική μεταβλητή που οι δημιουργοί θα αυξάνουν κατά 1 ενώ ο καταστροφέας θα την μειώνει κατά 1 (θα πρέπει να γράψουμε έναν καταστροφέα που θα κάνει μόνον αυτήν τη δουλειά). Αυτό όμως είναι πολύ άκομψο και καθόλου ασφαλές: μια καθολική μεταβλητή που έχει σχέση αποκλειστικά με την κλάση! Η C++ μας δίνει τη δυνατότητα να κάνουμε κάτι καλύτερο.

Μπορούμε να δηλώσουμε μια *στατική* μεταβλητή-μέλος με όνομα *objCnt* και της δίνουμε αρχική τιμή έξω από τη δήλωση της κλάσης:

```
class Date
{
public:
// ...
private:
   static size_t objCnt;
   unsigned int  dYear;
// ...
}; // Date

   size_t Date::objCnt = 0;
```

Δηλώνουμε έναν καταστροφέα:

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

και αλλάζουμε τους δημιουργούς:

```
Date::Date()
{
   dYear = 1;  dMonth = 1;  dDay = 1;
                                  // 1η Ιανουαρίου του έτους 1 μ.Χ.
   ++objCnt;
} // Date::Date

Date:: Date( int yp, int mp, int dp )
{
   if ( yp <= 0 )
      throw DateXptn( "Date", DateXptn::yearErr, yp );
   dYear = yp;
   if ( mp <= 0 || 12 < mp )
      throw DateXptn( "Date", DateXptn::monthErr, mp );
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
   ++objCnt;
} // Date::Date

Date::Date( const Date& d )
{
   dYear = d.dYear;  dMonth = d.dMonth;  dDay = d.dDay;
   ++objCnt;
```

[19] Άλλοι είναι πιο «κάθετοι». Στο (Lockheed-Martin 2005) ο *AV Rule* 177 λέει: *"User-defined conversion functions should be avoided"* δηλαδή: συναρτήσεις μετατροπής (τύπου) που ορίζονται από τον χρήστη πρέπει να αποφεύγονται!

```
} // Date::Date
```

Ακόμη, δηλώνουμε μια ανοικτή μέθοδο:

```
   static int getObjCnt()  {  return objCnt;  };
```

Αν τώρα δώσουμε:

```
   Date k1, k2, k3[5];
   cout << Date::getObjCnt() << endl;
   {
      Date s1( 1950, 2, 1 ), s2( 2000, 2, 3 ),
           s3( 1960, 3, 1 ), *c;
      cout << Date::getObjCnt() << endl;
      c = new Date [3];
      cout << Date::getObjCnt() << endl;
      delete [] c;
      cout << Date::getObjCnt() << endl;
   }
   cout << Date::getObjCnt() << endl;
```

θα πάρουμε:

```
7
10
13
10
7
```

Στην αρχή έχουμε 7 αντικείμενα κλάσης *Date*: πέντε του *k*3 και άλλα 2 από τις *k*1, *k*2. Στη συνέχεια, στη σύνθετη εντολή δηλώνουμε 3 αντικείμενα κλάσης *Date*: *s*1, *s*2, *s*3 και τα αντικείμενα γίνονται 10. Όταν πάρουμε με τον **new** άλλα 3 αντικείμενα έχουμε συνολικά 13. Στη συνέχεια ανακυκλώνουμε τη μνήμη (**delete**) που πήραμε για τον *c* και τα αντικείμενα ξαναμένουν 10. Στο τέλος της σύνθετης εντολής καλείται αυτομάτως ο κατάστροφέας και καταστρέφει τα *s*1, *s*2, *s*3· έτσι ξαναμένουμε με 7 αντικείμενα.

Ας κάνουμε τώρα μερικές παρατηρήσεις στο παράδειγμά μας:

1. Το (στατικό) μέλος *objCnt* δηλώνεται στην κλάση και παίρνει αρχική τιμή έξω από αυτήν.
2. Όπως καταλαβαίνεις από τον χειρισμό του, το *objCnt* είναι μέλος της κλάσης και όχι του κάθε αντικειμένου της.
3. Έτσι όταν αναφερόμαστε στο *objCnt* χρησιμοποιούμε επίλυση εμβέλειας ("**Date::objCnt**") και όχι επιλογή μέλους ("**.objCnt**").
4. Η μέθοδος *getObjCnt*, που χειρίζεται μόνον το μέλος *objCnt*, είναι, και αυτή, *στατική*. Και αυτή έχει σχέση με ολόκληρη την κλάση και, όπως βλέπεις αναφερόμαστε και σε αυτήν με το πρόθεμα "**Date::**".

- ***Γενικώς με το* `static` *δηλώνουμε μεταβλητές, μεθόδους ή και σταθερές που ανήκουν σε ολόκληρη την κλάση και όχι στο κάθε αντικείμενο ξεχωριστά.***

Και σταθερές; Ναι! Όπως ακριβώς δηλώσαμε το στατικό μέλος *objCnt* μπορούμε να δηλώσουμε και να δώσουμε τιμή σε σταθερά (**const**) μέλη οποιουδήποτε τύπου. Θα το δούμε στην επόμενη παράγραφο.

Στην §19.1.2 λέγαμε για τις βοηθητικές συναρτήσεις «*αργότερα ... θα τις διακοσμήσουμε και με ένα* "**static**"». Πράγματι, οι *lastDay()* και *isLeapYear()* επεξεργάζονται τα δεδομένα που τους περνούμε μέσω των παραμέτρων τους χωρίς να έχουν οποιαδήποτε κατ' ευθείαν σχέση με τα (μη στατικά) μέλη *dYear*, *dMonth*, *dDay*. Το ίδιο ισχύει και για τη *cStrLen* της *BString*. Από εδώ και πέρα, όλες αυτές θα τις δηλώνουμε "**static**"· το ίδιο θα κάνουμε και για όλες τις βοηθητικές συναρτήσεις που θα γράφουμε.

Αν ξαναγυρίσουμε στην Άσκ. 19-3 τώρα μπορούμε να δώσουμε την πιο καλή λύση για την *isValidDate()*· μια στατική συνάρτηση:

```
   static bool isValidDate( int ay, int am, int ad )
   {
      bool fv( true );
      try{  Date( ay, am, ad );  }
```

```
      catch( DateXptn& ) {  fv = false;  }
      return fv;
   } // isValidDate
```

που τη χρησιμοποιούμε κάπως έτσι:

```
   if ( Date::isValidDate(iy, im, id) ) . . .
```

21.15 «Σταθερά» Μέλη Κλάσης

Όπως είπαμε στην §19.1.1, η C++ επιτρέπει να βάλουμε το χαρακτηριστικό "`const`" σε ορισμένες μεθόδους. Τι σημαίνει; *Η μέθοδος που έχει το χαρακτηριστικό* `const` *δεν αλλάζει τις τιμές των μελών*. Για παράδειγμα, στη *BString* είχαμε δηλώσει:

```
// . . .
   BString& operator=( const BString& rhs );
   inline const char* c_str() const;
   inline size_t length() const;
   inline bool empty() const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   int compare( const BString& rhs ) const;
// . . .
```

αφού οι *c_str()*, *length()*, *empty()* και *compare()* δεν μεταβάλλουν τις τιμές μελών.

Γιατί να δηλώσουμε `const` κάποια μέθοδο;

- Για λόγους τεκμηρίωσης. Χωρίς να εξετάσουμε το τι κάνει η μέθοδος καταλαβαίνουμε ότι δεν μεταβάλλει τις τιμές των μελών.
- Για λόγους ασφάλειας. Αν προσπαθήσουμε να μεταβάλλουμε την τιμή κάποιου μέλους, ο μεταγλωττιστής θα μας επαναφέρει στην τάξη.

Μπορείς να δηλώσεις και σταθερά αντικείμενα, π.χ.:

```
   const BString aSt( "qwdfvbnh" );
```

Αν προσπαθήσεις να χειριστείς ένα τέτοιο αντικείμενο με μέθοδο που δεν είναι `const`, π.χ.:

```
   aSt.swap( bSt );
```

θα πάρεις μήνυμα λάθος από τον μεταγλωττιστή.

Υπάρχουν περιπτώσεις που θέλουμε να αλλάζουν οι τιμές ορισμένων μελών ακόμη και σε σταθερά αντικείμενα. Αν δηλώσεις τα μέλη αυτά ως "`mutable`" (μεταλλάξιμα) έχεις αυτήν τη δυνατότητα.

Μπορούμε να έχουμε μέλη-σταθερές; Ο πιο απλός τρόπος είναι αυτός που χρησιμοποιούμε στις κλάσεις εξαιρέσεων: δηλώνουμε μέσα στην κλάση έναν απαριθμητό τύπο με όσες σταθερές θέλουμε, π.χ.:

```
struct DateXptn
{
   enum { yearErr, monthErr, dayErr, outOfLimits };
// . . .
}; // DateXptn
```

και χρησιμοποιούμε στο πρόγραμμα τις σταθερές *yearErr*, *monthErr*, *dayErr*, *outOfLimits* ως: `DateXptn::yearErr`, `DateXptn::monthErr`, `DateXptn::dayErr`, `DateXptn::outOfLimits`.

Πρόσεξε τώρα το παρακάτω παράδειγμα για να δεις τι άλλες δηλώσεις σταθερών μπορείς να κάνεις:

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

using namespace std;

class A
{
private:
```

```
// . . .
   static const int size = 10;
public:
// . . .
   static const double cx;
   static const string car;
   static const char   carc[size];
}; // A
   const double A::cx = 1.234;
   const string A::car = "123456789";
   const char  A::carc[A::size] = "123456789";

int main()
{
   cout << A::cx << "  " << A::car << "  " << A::carc << endl;
}
```

Μέσα στην κλάση μπορείς να ορίσεις *ακέραιες* σταθερές, όπως η *size*.

Σταθερές άλλων τύπων, όπως οι *cx*, *car*, *carc*, δηλώνονται μέσα στην κλάση αλλά ορίζονται έξω από αυτήν.

Όπως καταλαβαίνεις, με βάση τα παραπάνω, μπορούμε να γράψουμε:

```
class BString
{ // I: (0 <= bsLen < bsReserved) && (bsReserved % bsIncr == 0)
public:
// ...
private:
   static const size_t bsIncr = 16;
   char*  bsData;
   size_t bsLen;
   size_t bsReserved;
}; // BString
```

21.16 «Σταθερά» Μέλη Αντικειμένου

Καλά τα σταθερά μέλη κλάσης έχουν νόημα· τα *σταθερά μέλη ενός μη-σταθερού αντικειμενου* τι νόημα έχουν; Ως απάντηση, θα ξαναθυμίσουμε αυτά που λέγαμε στο παράδειγμα της μπαταρίας (§19.7) «*δεν θα πρέπει να υπάρχουν μέθοδοι που να αλλάζουν τις τιμές των bVoltage και bMaxEnergy, αφού αυτά παριστάνουν σταθερά χαρακτηριστικά της μπαταρίας που της αποδίδονται με την κατασκευή* (*δημιουργία*) *της*.» Και ακόμη καλύτερο θα ήταν να δηλώναμε:

```
class Battery
{
public:
// . . .
private:
   const double bVoltage;   // volts
   const double bMaxEnergy; // joules
   double       bEnergy;    // joules
}; // Battery
```

Ωραία! Αυτό επιτρέπεται, αλλά *πώς θα δίνουμε τιμές στα bVoltage και bMaxEnergy του κάθε αντικειμένου που δηλώνουμε*; Από τη στιγμή που δημιουργούνται τα μέλη απαγορεύεται να αλλάξουμε την (όποια) τιμή τους Ο μόνος τρόπος είναι η *λίστα εκκίνησης* που ορίζει τις τιμές τους όταν δημιουργούνται. Και πάλι δηλώνουμε:

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

αλλά τώρα ορίζουμε:

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

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

Και στην κλάση για την πισίνα, οι διαστάσεις της πισίνας θα πρέπει να είναι σταθερές. Αφήνουμε ως άσκηση τις σχετικές αλλαγές.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

21-1 Όπως ξέρεις μπορούμε να δηλώσουμε:

```
   string os1( 7, '#' );
```

και η *os1* να πάρει ως τιμή το "**#######**". Εφοδίασε τη *BString* με το εργαλείο που χρειάζεται για να αποκτήσει αυτήν τη δυνατότητα.

Β Ομάδα

21-2 Γράψε μια μέθοδο *setToday()* για τη *Date* ώστε να δίνει στο αντικείμενο ως τιμή την τρέχουσα ημερομηνία από το ρολόι του υπολογιστή.

21-3 Θέλουμε ένα εργαλείο, ας το πούμε *today()*, της *Date* που θα μας δίνει την τρέχουσα ημερομηνία. Πώς θα μπορούσε να γίνει; Σκέψου το σε σχέση με το εργαλείο της προηγούμενης άσκησης αλλά και ανεξάρτητα από εκείνο.