κεφάλαιο

22

Επιφόρτωση Τελεστών

Ο στόχος μας σε αυτό το κεφάλαιο:
Να γνωρίσουμε μερικούς κανόνες για την επιφόρτωση τελεστών σε κλάσεις. Αυτοί είναι κάπως διαφορετικοί από αυτούς που ξέρουμε μέχρι τώρα· οι αλλαγές είναι απαραίτητες αφού στις κλάσεις έχουμε και κρυμμένα μέλη. Η δυνατότητα για επιφόρτωση με μεθόδους αξιοποιείται στον μέγιστο βαθμό.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να επιφορτώνεις οποιονδήποτε τελεστή σε οποιαδήποτε κλάση και να χρησιμοποιείς μερικές πάγιες τεχνικές επιφόρτωσης. Θα μπορείς να αποφασίσεις αν είναι σωστό να γίνει κάποια επιφόρτωση.

Έννοιες κλειδιά:

- *επιφόρτωση τελεστή*
- *συναρτήσεις* ***friend***
- *κλάσεις* ***friend***
- *προσεγγιστές*
- *τεχνική* ***pimpl***

Περιεχόμενα:

Εισαγωγικές Παρατηρήσεις:

Είναι πολύ βολικό να επιφορτώνεις τελεστές, για μια κλάση που γράφεις, αντί να γράφεις συναρτήσεις. Ή, αν θέλεις, να επιφορτώνεις *και* τελεστές εκτός από τις συναρτήσεις. Ο λόγος;

- Όπως είδαμε ήδη με τις δομές, μπορείς να χρησιμοποιήσεις για τα αντικείμενα περιγράμματα συναρτήσεων (ή και κλάσεων, όπως θα δούμε στη συνέχεια). Π.χ. για να χρησιμοποιήσεις περίγραμμα συνάρτησης για αναζήτηση σε πίνακα με στοιχεία τύπου *T* συνήθως απαιτείται να έχουμε επιφορτώσει για τον *T* με τον τελεστή "`==`", ενώ για να χρησιμοποιήσεις περίγραμμα συνάρτησης για ταξινόμηση απαιτείται τουλάχιστον ο "`<`".
- Θυμάσαι τι κάνουν! Αντί να ψάχνεις μέσα στην κλάση να βρεις αν τη μέθοδο την έχεις πει *next* ή *nextDay* ή *epomenh* ή *epomeni* ή *epom* χρησιμοποιείς τον "`++`" και τελείωσες! Αυτό βέβαια με μια προϋπόθεση:

♦ ***Όταν επιφορτώνεις έναν τελεστή για κάποιον δικό σου τύπο, η δράση του θα πρέπει να είναι παρόμοια με αυτήν που ήδη είναι γνωστή από τους πρωτογενείς τύπους.***

Για πρώτη φορά μιλήσαμε για επιφόρτωση τελεστών στην §14.6 ενώ στα κεφάλαια 20 και 21 ασχοληθήκαμε διεξοδικώς με την επιφόρτωση του τελεστή εκχώρησης "`=`".

Στο κεφάλαιο αυτό, που ασχολείται με την επιφόρτωση τελεστών για κλάσεις, θα συμπληρώσουμε (και θα τροποποιήσουμε) τις οδηγίες της §14.6.4.

22.1 Επιφόρτωση Τελεστών: Τι Ξέρουμε Μέχρι Τώρα

Όπως έχουμε μάθει, μπορούμε να επιφορτώσουμε οποιονδήποτε τελεστή εκτός από τους

"`::`", "`.`", "`.*`", "`?:`"

Ο τρόπος που μάθαμε να κάνουμε την επιφόρτωση ήταν ένας: μια καθολική συνάρτηση με το όνομα "`operator@`" για τον τελεστή "`@`". Αλλά για δες πώς επιφορτώσαμε τον "`==`" στην §15.5.1:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day == rhs.day );
}; // operator==( const Date
```

και πώς στην §19.1.4:

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

Τι μεσολάβησε; Στο κεφάλαιο 19 «κρύψαμε» τα μέλη της *Date* σε περιοχή **private**! Είπαμε βέβαια ότι οι "*get*" δεν αυξάνουν τον υπολογιστικό χρόνο αφού αυτές οι μέθοδοι είναι **inline**. Τι γίνεται όμως στις περιπτώσεις που θα χρειαστούμε μέλη για τα οποία δεν γράφουμε τέτοιες μεθόδους;

Και κάτι ακόμη: Ας πούμε ότι, για να προχωρούμε στην επόμενη μέρα, θέλουμε να επιφορτώσουμε τον "`++`" για τη *Date*. Θα χρειαστούμε τη *lastDay()*· αλλά και αυτήν την «κρύψαμε» σε περιοχή **private**.

Υπάρχει λοιπόν πρόβλημα με τα «κρυμμένα» μέλη και οι λύσεις –εκτός από τις "*get*"– είναι δύο:

- Να επιφορτώσουμε τους τελεστές με μεθόδους.
- Να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου στην (καθολική) συνάρτηση που επιφορτώνει κάποιον τελεστή.

Θα δούμε και τις δύο περιπτώσεις. Αλλά, όπως θα δεις στην επόμενη παράγραφο, η «επιλεκτική αποκάλυψη μυστικών» είναι μια, κατ' αρχήν τουλάχιστον, επικίνδυνη τεχνική. Θα προσπαθούμε λοιπόν να κάνουμε τις επιφορτώσεις μας με μεθόδους ή με καθολικές συναρτήσεις που χρησιμοποιούν συναρτήσεις-μέλη "*get*" που είναι **inline**.

Είδαμε ήδη την επιφόρτωση τελεστή με μέθοδο: αυτήν του "`=`" για τη *BString*. Η επιφόρτωση έγινε με μια μέθοδο:

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

Όταν τη γράφαμε, παίρναμε υπόψη μας ότι στην εκχώρηση "`a = Π`" το *Π* γίνεται *rhs* ενώ `*this` είναι το *a*. Αν δηλαδή είχαμε υλοποίηση με καθολική συνάρτηση[1], σύμφωνα με τις αντιστοιχίες που είδαμε στην §19.1 και 19.1.1 θα είχαμε:

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

Ο δυαδικός τελεστής "`=`" έχει τα εξής χαρακτηριστικά:

- *Έχει αριστερά το αντικείμενο* που μας ενδιαφέρει (`*this`).
- *Δεν είναι αντιμεταθετικός*, δηλαδή δεν είναι δυνατόν να γράψουμε "`Π = a`" αντί για "`a = Π`".

Οποιονδήποτε τελεστή με αυτά τα χαρακτηριστικά (άλλα παραδείγματα: "`+=`", "`[]`" κλπ) τον επιφορτώνουμε με μέθοδο.

Με μέθοδο επιφορτώνουμε κάι κάθε ενικό τελεστή.

Αν ο τελεστής είναι *δυαδικός αντιμεταθετικός* η επιφόρτωση με μέθοδο δεν είναι καλή ιδέα. Ας πάρουμε τον "==" για τη *BString*: Αν τον επιφορτώσεις με μέθοδο:

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

η σύγκριση "`a == "qwerty"`" θα είναι εφικτή ενώ η "`"qwerty" == a`" είναι παράνομη αφού το `"qwerty"` δεν είναι τιμή *BString*. Εδώ η επιφόρτωση θα πρέπει να γίνει όπως ξέρουμε: με καθολική συνάρτηση.

Με καθολική συνάρτηση επιφορτώνονται και οι *δυαδικοί μη αντιμεταθετικοί* τελεστές με το αντικείμενο στο *δεξιό μέρος*.

22.2 Προβλήματα Συμβατότητας

Στην §14.6, μιλώντας για επιφόρτωση τελεστών για πρώτη φορά βάζαμε έναν βασικό κανόνα:

♦ ***Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.***

πράγμα που λέμε με άλλα λόγια και στην εισαγωγή του παρόντος κεφαλαίου.

Η τήρηση αυτού του κανόνα είναι απαραίτητη για να διευκολυνόμαστε από τις επιφορτώσεις. Η παράβασή του αντί για διευκόλυνση προκαλεί σύγχυση και προβλήματα στον προγραμματιστή-χρήστη της κλάσης.

Το επόμενο πρόβλημα είναι αυτό που ήδη αντιμετωπίσαμε με τον τελεστή "`=`" για τη *BString* και τη μέθοδο *assign()*. Για την επιφόρτωση του "`=`" η γλώσσα επιβάλλει μέθοδο με επικεφαλίδα:

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

[1] *Ο μεταγλωττιστής δεν θα σου επιτρέψει* να επιφορτώσεις τον "`=`" με άλλον τρόπο

Η *assign()*, που θα πρέπει να έχει ακριβώς την ίδια λειτουργία θα πρέπει –κατά παράβαση άλλων κανόνων που έχουμε θέσει– να έχει ακριβώς την ίδια επικεφαλίδα:

```
BString& assign( const BString& rhs );
```

Και μετά; Αντιγράφουμε το σώμα της πρώτης συνάρτησης ως σώμα της δεύτερης; Όπως είπαμε, η αντιγραφή είναι προσωρινή μόνον εγγύηση ταυτόσημης λειτουργίας. Η ταυτόσημη λειτουργία μπορεί να χαθεί όταν, αργότερα, θα χρειαστεί να τροποποιήσεις τις μεθόδους.

Η μοναδική εγγύηση είναι το να γράψουμε τον κώδικα μια φορά μόνον –είτε στην `operator=()` είτε στην *assign()*– και η δεύτερη επιφόρτωση να καλεί την πρώτη. Στην §20.4.1 ορίσαμε την `operator=` και ορίσαμε –**inline**– την *assign()* ως:

```
BString& assign( const BString& rhs ) { return (*this = rhs); }
```

Έτσι, αν έχουμε δηλώσει:

```
BString a, b;
```

η εντολή:

```
b.assign( a );
```

θα μετατραπεί σε "`b = a`".

Αν προτιμούσαμε να ορίσουμε την *assign()*, θα έπρεπε να ορίσουμε –και πάλι **inline**–

```
BString& operator=( const BString& rhs )
{  return ( this->assign(rhs) );  }
```

Σε μια τέτοια περίπτωση, η εντολή:

```
b = a;
```

θα μετατραπεί σε "`b.assign( a )`".

Είναι φανερό ότι –και στις δύο περιπτώσεις– το πρόγραμμά μας δεν «πληρώνει» οποιοδήποτε κόστος κλήσης συνάρτησης.

Όπως θα δεις στη συνέχεια, με τον ίδιο τρόπο θα αντιμετωπίσουμε το ζεύγος "`+=`", *append()* –για τη *BString*– και το ζεύγος "`+=`", *forward()*, για τη *Date*.

Γενικώς λοιπόν:

- ***Αν θέλουμε δύο μέθοδοι, ας πούμε a και b, να εκτελούν την ίδια ακριβώς λειτουργία ορίζουμε τη μια από αυτές –έστω την a– και μετά ορίζουμε τη b με κλήση της a.***

Ένα άλλο πρόβλημα συμβατότητας είναι αυτό των τελεστών με σχετικό νόημα, π.χ.: "`+`" και "`+=`". Αυτοί οι δύο τελεστές επιφορτώνονται για τη *BString* ώστε να κάνουν –όπως και στη *std*`::`*string*– σύνδεση δύο ορμαθών. Η "`a + b`" μας δίνει έναν νέο ορμαθό που αποτελείται από το κείμενο του *a* και στη συνέχεια το κείμενο του *b* χωρίς να αλλάζουν οι τιμές των *a* και *b*. Η "`a += b`" επισυνάπτει στο κείμενο του *a* το κείμενο του *b* αλλάζοντας την τιμή του *a*. Όπως θα δεις, η επιφόρτωση του "`+`" γίνεται με κλήση του "`+=`"· δηλαδή ο ορισμός της πράξης γράφεται μια φορά μόνον, για τον "`+=`".

Με τους "`+`" και "`-`" το πρόβλημα μπορεί να είναι ευρύτερο. Για παράδειγμα, για τη *Date* επιφορτώνουμε τον "`+`"

`+`: *Date* × `int` ↛ *Date*

με το εξής νόημα: *η "*`d1 = d + n`*" δίνει στη d1 ως τιμή ημερομηνία μεταγενέστερη κατά n μέρες της τιμής της d.* Αν τώρα θέλουμε να επιφορτώσουμε τον "`+=`" θα πρέπει να το κάνουμε έτσι που η "`d += n`" να έχει το ίδιο νόημα με την "`d = d + n`"· οτιδήποτε άλλο θα ήταν παραπλανητικό για τον προγραμματιστή-χρήστη της κλάσης. Τα ίδια ισχύουν και για τον "`++`": το "`++d`" δεν μπορεί παρά να σημαίνει "`d = d + 1`" ή "`d += 1`". Όπως θα δεις στη συνέχεια, ορίζουμε αρχικώς τη *forward*, στη συνέχεια επιφορτώνουμε τον "`+=`" ως ταυτόσημο με αυτήν και από τον "`+=`" ορίζουμε τις επιφορτώσεις των "`+`" και "`++`". Ο μεταθεματικός "`++`" επιφορτώνεται με κλήση του προθεματικού. Η άσκ. 22-1 σου ζητάει να συνεχίσεις, με την ίδια λογική, με τους "`-`", "`--`" και "`-=`".

- ***Αν έχεις να επιφορτώσεις για μια κλάση τους "+", "+=", "++", "-", "-=", "—" όρισε έναν από αυτούς και όρισε με κλήσεις σε αυτόν τις επιφορτώσεις των άλλων.***

Παρόμοια ισχύουν και τους "`*`", "`*=`","`/`", "`/=`" κ.ο.κ.

Θα πει κάποιος: «Και τι θα γίνει αν εγώ επιφορτώνω τον κάθε τελεστή με όποιο νόημα μου έλθει;» Δεδομένου του δημοκρατικού πολιτεύματος στο οποίο ζούμε, είναι σίγουρο ότι δεν θα πάει στη φυλακή! Αυτό που δεν είναι σίγουρο είναι ότι θα βρίσκει πελάτες για να πουλάει το λογισμικό που γράφει...

22.3 Συναρτήσεις και Κλάσεις "`friend`"

Ας δούμε τώρα πώς μπορούμε να κάνουμε «επιλεκτική αποκάλυψη μυστικών» ενός αντικειμένου σε μια καθολική συνάρτηση.

Η C++ μας δίνει όμως την εξής δυνατότητα: να δηλώσουμε μέσα σε μια κλάση, ας πούμε τη *Date*, ως **φίλη** (friend) μια συνάρτηση (τελεστή), ας πούμε την `operator==()`:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs );
public:
   Date();
// . . .
```

Με ποιο αποτέλεσμα;

- ***Αν μια συνάρτηση δηλωθεί ως `friend` (φίλη) μέσα σε μια κλάση, έχει τη δυνατότητα να «βλέπει» τα εσωτερικά μέλη των αντικειμένων της κλάσης.***

Έτσι, μπορούμε να γράψουμε, όπως παλιά:

```
bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
```

Με αυτόν τον τρόπο γλυτώσαμε από κλήσεις συναρτήσεων. Για την ακρίβεια δεν τις βλέπουμε πια· διότι –όπως είπαμε– και οι κλήσεις των "*get*" εξαφανίζονταν από τον μεταγλωττιστή λόγω του "`inline`".

Για μικρές καθολικές συναρτήσεις μπορείς, αν θέλεις, να βάλεις ολόκληρο τον ορισμό μαζί με τη δήλωση:

```
class Date
{
friend bool operator==( const Date& lhs, const Date& rhs )
{
   return ( lhs.dYear == rhs.dYear && lhs.dMonth == rhs.dMonth &&
            lhs.dDay == rhs.dDay );
}; // operator==( const Date
public:
   Date();
// . . .
```

Στην περίπτωση αυτήν η συνάρτηση (στο παράδειγμά μας: `operator==()`) ορίζεται "`inline`".

Εκτός από συναρτήσεις είναι δυνατόν να δηλώσουμε και *φίλες κλάσεις*. π.χ.:

```
class A
{
friend class B;
public:
// ...
```

Όλες οι μέθοδοι ενός αντικειμένου κλάσης *B* που δηλώνεται ως "`friend`" μέσα στην κλάση *A*, έχουν τη δυνατότητα να βλέπουν τα εσωτερικά μέλη των αντικειμένων της κλάσης *A*. Αλλά:

- ***Η φιλία δεν είναι μεταβατική.***

Δηλαδή: το ότι η κλάση *B* δηλώνεται ως "`friend`" της *A* και μια συνάρτηση ή κλάση δηλώνεται ως "`friend`" της *B* δεν σημαίνει ότι είναι και φίλη της *A*.

Όπως καταλαβαίνεις, αυτό το «άνοιγμα» των «μυστικών» ενός αντικειμένου σε συναρτήσεις και –πολύ περισσότερο– σε κλάσεις δεν είναι ακίνδυνο. Όπως σου λέει και η πείρα σου, μερικές φορές φτάνει ένα "`==`" να γίνει "`=`" για να γίνει το «κακό». Για τον λόγο αυτόν θα χρησιμοποιούμε τις φιλίες όσο γίνεται λιγότερο και με τη μεγαλύτερη δυνατή προσοχή.

Ειδικώς για τις φιλίες με κλάσεις, ο κίνδυνος μπορεί να μετριασθεί αν δηλώσεις "`friend`" όχι ολόκληρη την κλάση αλλά συγκεκριμένη (-ες) μέθοδο (-ους) της. Ας πούμε ότι έχεις:

```
class Date;

class K
{
public:
   K( int aa=5 )  {  a = aa;  };
   int f( const Date& d ) const;
   int g( const Date& d ) const;
private:
   int a;
}; // K
```

και δηλώσεις:

```
class Date
{
friend int K::f( const Date& d ) const;
//  . . .
```

μπορείς στον ορισμό της *K*`::`*f* να χρησιμοποιείς τα κρυμμένα μέλη της παραμέτρου *d* ενώ αυτό απαγορεύεται στον ορισμό της *K*`::`*g*.

Εκείνο το "`class Date;`" στην αρχή τί είναι; Διάβασε παρακάτω...

22.4 Προειδοποιητική Δήλωση

Για να μπορέσουμε να κάνουμε τη δήλωση

```
class K
{
// . . .
   int f( const Date& d ) const;
// . . .
```

στο προηγούμενο παράδειγμα θα πρέπει ο μεταγλωττιστής να ξέρει τη *Date*. Δεν χρειάζεται να την ξέρει πλήρως· αρκεί μια **προειδοποιητική δήλωση** (forward declaration)

```
   class Date;
```

που –μοιάζει με τις επικεφαλίδες συναρτήσεων που βάζουμε πριν από την κλήση τους και– λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

22.5 Ενικοί Τελεστές

Δύο από τις οδηγίες της §14.6.4 που θα αλλάξουμε είναι αυτές για την *επιφόρτωση ενικών τελεστών για κλάσεις*.

- ***Οι ενικοί τελεστές για τις κλάσεις θα επιφορτώνονται με μεθόδους.***

22.5.1 Προθεματικοί Ενικοί Τελεστές

Λέγαμε στην §14.6.4:

- «*Επιφορτώνουμε έναν προθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

  Τώρα συμπληρώνουμε, αν *C* είναι μια κλάση:

- Αν ο *T* είναι *C*`&` η επιφόρτωση μπορεί θα γίνεται με μέθοδο (χωρίς παραμέτρους):

  `Trv operator@( )`

  Αν ο *T* είναι *C* ή `const` *C*`&` η επιφόρτωση μπορεί θα γίνεται με μέθοδο:

  `Trv operator@( ) const`

Παράδειγμα

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον *προθεματικό* "`++`" για την *Date*. Τι θα κάνει; Θα προχωράει την ημερομηνία κατά μια ημέρα. Μόνον αυτό; Για να είμαστε συνεπείς με τη φιλοσοφία της C++ (C) θα πρέπει να επιστρέφει τη νέα ημερομηνία. Σύμφωνα με τις οδηγίες της §14.6.4 θα έπρεπε η επιφόρτωση να είναι της μορφής:

```
Date& operator++( Date& a )
```

αλλά σύμφωνα με την παραπάνω συμπλήρωση θα πρέπει να γίνει με μέθοδο:

```
Date& Date::operator++()
{
   if ( dDay < lastDay(dYear, dMonth) )
      ++dDay;
   else // αλλαγή μήνα
   {
      dDay = 1;
      if ( dMonth < 12 )
         ++dMonth;
      else // αλλαγή έτους
      {
         dMonth = 1;
         ++dYear;
      } // if ( dMonth...
   } // if ( dDay...
   return *this;
} // Date::operator++
```

Πως τον χρησιμοποιούμε;

```
Date d1( 2011, 3, 31 );
cout << d1 << endl;
++d1;
cout << d1 << endl;
```

Αντί για "`++d1`" μπορείς να γράψεις:

```
d1.operator++();
```

αλλά γιατί να το κάνεις;

Αν θελήσεις να επιφορτώσεις τον "`++`" με καθολική συνάρτηση (με μια παράμετρο) θα πρέπει να τη δηλώσεις "`friend`" οπωσδήποτε για να μπορέσεις να χρησιμοποιήσεις τη *lastDay*, που είναι **private**.

Παρατήρηση: ▶

Μετά αυτά που είπαμε στην §22.2, καταλαβαίνεις ότι με τον "`++`" της *Date* δεν τελειώσαμε...◀

Αργότερα θα επιφορτώσουμε και άλλους προθεματικούς ενικούς τελεστές.

22.5.2 Ενικοί Μεταθεματικοί Τελεστές

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν μεταθεματικό ενικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( T a, int )`

  όπου Trv είναι ο τύπος του αποτελέσματος, T ο τύπος της παραμέτρου.»

και τώρα συμπληρώνουμε: αν *C* είναι μια κλάση

- Αν ο *T* είναι *C*`&` η επιφόρτωση μπορεί να γίνει με μέθοδο της *C*:

  `Trv operator@( int )`

  Αν ο *T* είναι *C* ή `const` *C*`&` η επιφόρτωση μπορεί να γίνει με μέθοδο:

  `Trv operator@( int ) const`

Παράδειγμα

Ας πούμε ότι θέλουμε να επιφορτώσουμε τον μεταθεματικό "`++`" για τη *Date*. Ο *T* σίγουρα είναι ο *Date*`&`. Επομένως συζητούμε για την περίπτωση:

`Trv operator++( int )`

Ο *Trv* ποιος θα είναι; Ο *Date* ή ο *Date*`&`; Ο τελεστής αυτός αλλάζει την τιμή του αντικειμένου αλλά επιστρέφει την προηγούμενη τιμή του. Αυτή θα πρέπει να έχει κρατηθεί σε μια *τοπική μεταβλητή της συνάρτησης*. Άρα δεν είναι δυνατόν να επιστρέψει τιμή τύπου *Date*`&`. Επομένως θα έχουμε:

`Date operator++( int )`

Κατά τα άλλα, η υλοποίηση είναι πολύ απλή αν χρησιμοποιήσουμε τον προθεματικό "`++`":

```
Date Date::operator++( int )
{
   Date sv( *this );
   ++(*this);
   return sv;
} // Date::operator++(int)
```

22.6 Μη-Αντιμεταθετικοί Δυαδικοί Τελεστές

Λέμε ότι ένας δυαδικός τελεστής είναι μη-αντιμεταθετικός όταν διαχείριζόμαστε με διαφορετικό τρόπο τα δύο μέρη του. Συνήθως αυτό συνοδεύεται και από διαφορά τύπου (τουλάχιστον κατά ένα "`const`").

22.6.1 Αντικείμενο Αριστερά

Λέγαμε στην §14.6.4:

- *«Επιφορτώνουμε έναν δυαδικό τελεστή* `@` *με μια συνάρτηση*

  `Trv operator@( Tl a, Tr b )`

  όπου Trv είναι ο τύπος του αποτελέσματος, Tl ο τύπος της πρώτης παραμέτρου και Tr ο τύπος της δεύτερης παραμέτρου.»

  Τώρα συμπληρώνουμε: Αν έχουμε το αντικείμενο αριστερά, αν δηλαδή ο *Tl* είναι *C*`&` ή *C* ή `const` *C*`&` η επιφόρτωση γίνεται με μέθοδο της κλάσης *C* και πιο συγκεκριμένα:

- Αν ο *Tl* είναι *C*`&` με μέθοδο με μια παράμετρο:

  `Trv operator@( Tr b )`

  Αν ο *Tl* είναι *C* ή `const` *C*`&` με μέθοδο:

  `Trv operator@( Tr b ) const`

Ως παράδειγμα θα επιφορτώσουμε τον τελεστή "`[]`" για τη *BString* και τον "`+=`" για τη *BString* και για τη *Date*.

22.6.1.1 Ο Τελεστής "`[]`" για τη *BString*

Μετά τη δήλωση:

```
string q( "qazwsx" );
```

η εντολή:

```
cout << q << "  " << q[2] << endl;
```

δίνει:

```
qazwsx  z
```

Ακόμη οι:[2]

```
q[2] = 'a';
cout << q << endl;
```

δίνουν:

```
qaawsx
```

Για να είναι αυτό δυνατόν στη *BString*, ο τελεστής "`[]`" θα πρέπει να επιφορτώνεται ως:

```
char& operator[]( BString a, int b )
```

Πρόσεξε ότι επιστρέφει τιμή τύπου "`char&`" (τιμή-*l*) και όχι "`char`". Σύμφωνα με αυτά που είπαμε παραπάνω θα πρέπει να τον επιφορτώσουμε ως μέθοδο

```
char& operator[]( int pos ) const {  return bsData[pos];  }
```

Μετά από αυτό, αν έχουμε ορίσει:

```
   BString q( "qazwsx" );
```

οι εντολές:

```
   cout << q.c_str() << "  " << q[2] << endl;
   q[2] = 'a';
   cout << q.c_str() << endl;
```

θα δώσουν:

```
qaawsx
qaawsx  a
```

Το ίδιο αποτέλεσμα παίρνεις και με τις:

```
   cout << q.c_str() << "  " << q.operator[](2) << endl;
   q.operator[](2) = 'b';
   cout << q.c_str() << endl;
```

22.6.1.2 Ο Τελεστής "+=" για τη *BString*

Έστω ότι θέλουμε να χρησιμοποιήσουμε τον τελεστή "`+=`" για να συνδέσουμε στο τέλος ορμαθού έναν άλλο ορμαθό.[3] Η επιφόρτωση θα είναι της μορφής:

```
BString& operator+=( BString& lhs, const BString& rhs )
```

ώστε στο πρόγραμμά μας να μπορούμε να γράφουμε:

```
s0 += s1;
```

Σύμφωνα με αυτά που είπαμε θα πρέπει να τον επιφορτώσουμε με μια μέθοδο:

```
BString& BString::operator+=( const BString& rhs )
```

---

[2] Να υπενθυμίσουμε: για την *std*`::`*string* η διαφορά των *at* και "`operator[]`" είναι ότι
- η *at* κάνει έλεγχο του δείκτη ενώ
- ο "`operator[]`" τον αφήνει στον προγραμματιστή.

[3] Αν s_1, s_2 ορμαθοί, τότε η s_1 `+=` s_2 ισοδυναμεί με $s_1 \leftarrow s_1 ^\frown s_2$, όπου "$^\frown$" είναι ο τελεστής σύνδεσης ακολουθιών.

Το πρόγραμμα δεν είναι απλό. Ας δούμε τι περιπτώσεις υπάρχουν (είμαστε «μέσα στο *s0*» και *rhs* είναι το *s1*):

- Να μην χρειαστούμε επιπλέον μνήμη. Πότε συμβαίνει αυτό; Όταν *bsLen* + *rhs*.*bsLen* + 1 ≤ *bsReserved*. Στην περίπτωση αυτή όλα είναι απλά:

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

- Όταν *bsLen* + *rhs*.*bsLen* + 1 > *bsReserved* χρειαζόμαστε μνήμη.

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsLen + rhs.bsLen + 1 > bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsLen+rhs.bsLen+1)/bsIncr+1)*bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc )
      {  throw BStringXptn( "operator+=",
                            BStringXptn::allocFailed );  }
      for ( int k(0); k < bsLen; ++k )  tmp[k] = bsData[k];
      delete[] bsData;
      bsData = tmp;
   }
   for ( int j(0), k(bsLen); j < rhs.bsLen; ++j, ++k )
      bsData[k] = rhs.bsData[j];
   bsLen += rhs.bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις, αν δεν καταφέρουμε να πάρουμε μνήμη –και ριχτεί εξαίρεση– δεν θα αλλάξει η τιμή του αντικειμένου (ασφάλεια προς τις εξαιρέσεις επιπέδου ισχυρής εγγύησης).

Να και ένα παράδειγμα χρήσης:

```
      BString s2( "qazwsxedcrf" ), s3( "res" ), s4( "qwerty" );

      cout << s2.c_str() << "  " << s2.length() << endl
           << s3.c_str() << "  " << s3.length() << endl
           << s4.c_str() << "  " << s4.length() << endl;

      s4 += s3;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
      s4 += s2;
      cout << s4.c_str() << "  " << s2.c_str() << "  "
           << s3.c_str() << endl;
```

Αποτέλεσμα:

```
qazwsxedcrf  11
res  3
qwerty  6
qwertyres  qazwsxedcrf  res
qwertyresqazwsxedcrf  qazwsxedcrf  res
```

Μην αμφιβάλλεις: αν γράψεις "`s4.operator+=(s3)`" αντί για "`s4 += s3`" και "`s4.operator+=(s2)`" αντί για "`s4 += s2`" θα πάρεις τα ίδια αποτελέσματα ακριβώς.

Στη *string* υπάρχει και η μέθοδος *append* που κάνει τα ίδια ακριβώς με τον "`+=`". Για αυτήν τη μέθοδο θα πρέπει να παραβιάσουμε τους κανόνες μας (σύμφωνα με τους οποίους θα πρέπει να την κάνουμε "`void`"): την υλοποιούμε αντιγράφοντας την υλοποίηση του τελεστή και βάζοντας μόνον στην επικεφαλίδα "`append`" όπου "`operator+=`":

```
BString& BString::append( const BString& rhs )
// ΤΑ ΥΠΟΛΟΙΠΑ ΙΔΙΑ ΜΕ ΤΟΝ operator+=
```

ή, ακόμη καλύτερα, με τον ορισμό (**inline**)

```
  BString& append( const BString& rhs ) { return (*this += rhs); }
```

22.6.1.3 Ο Τελεστής "+=" για τη *Date*

Έστω ότι θέλουμε να γράψουμε μια μέθοδο για την κλάση *Date*, ας την πούμε *forword*, που θα προχωράει μια ημερομηνία. Δηλαδή, αν έχουμε δηλώσει:

```
   Date d( 2007, 10, 15 );
```

τότε η:

```
   d.forward( 5 );
```

θα προχωρήσει τη *d* κατά 5 ημέρες και η τιμή της θα γίνει 20.10.2007.

Σύμφωνα με τους κανόνες μας, θα πρέπει να τη δούμε ως "`forward(d, 5)`" όπου μεταβάλλεται η τιμή του πρώτου ορίσματος και να υλοποιήσουμε ως `void`.

Για την ίδια λειτουργία θα επιφορτώσουμε και τον τελεστή "`+=`", πράγμα πολύ φυσικό αφού το ίδιο κάνει και για τους ακέραιους. Εδώ όμως τα πράγματα αλλάζουν: Η "`d += 5`" θα πρέπει να είναι πράξη που θα επιστρέφει τη νέα τιμή της *d*. Επειδή γίνεται το ίδιο με τους αριθμούς; Ναι, αλλά και για τον εξής λόγο: έχουμε ήδη επιφορτώσει τον "`++`". Είναι επιθυμητό οι "`++d`" και "`d += 1`" να έχουν το ίδιο αποτέλεσμα. Η επιφόρτωση θα είναι της μορφής:

```
Date& operator+=( Date a, int dd )
```

Θα την υλοποιήσουμε ως μέθοδο

```
   Date& operator+=( long int dd );
```

Η *forword()* θα πρέπει να συμπεριφέρεται παρομοίως. Θα έχουμε λοιπόν:

```
   Date& forward( long int dd );
```

Ο πιο απλός τρόπος για να υλοποιήσουμε τις μεθόδους είναι να χρησιμοποιήσουμε τις συναρτήσεις διαχείρισης χρόνου της C. Στην επόμενη υποπαράγραφο θα δούμε εν συντομία αυτά που μας ενδιαφέρουν.

22.6.1.4 * Ο Χρόνος στη C

Για να χρησιμοποιήσεις τις συναρτήσεις διαχείρισης χρόνου σε ένα πρόγραμμα θα πρέπει να περιλάβεις το ctime:

```
#include <ctime>
```

Η πρώτη συνάρτηση που θα δούμε είναι η:

```
time_t time( time_t* timer );
```

που μας δίνει τα δευτερόλεπτα που έχουν περάσει από την 01.01.1970, ώρα 00:00:00 GMT[4]. Ο τύπος *time_t* είναι μετονομασία του `long int`.

Μπορείς να πάρεις τον τρέχοντα χρόνο στο πρόγραμμά σου δηλώνοντας:

```
   time_t t1;
```

και δίνοντας: "`t1 = time(0)`" ή "`time(&t1)`".

Ελάχιστη επιτρεπόμενη τιμή της *t1* είναι το 0, που αντιστοιχεί σε 01.01.1970, ώρα 00:00:00 GMT (ή 02:00:00 ώρα Ελλάδας). Μέγιστη επιτρεπόμενη τιμή είναι η `LONG_MAX` που αντιστοιχεί σε 19.01.2038, ώρα 03:14:07 GMT (ή 05:14:07 ώρα Ελλάδας).

Η συνάρτηση

```
tm* localtime( const time_t* timer );
```

μετατρέπει μια τιμή τύπου `time_t` σε τιμή τύπου:

```
struct tm
{
   int tm_sec;     // seconds
   int tm_min;     // minutes
```

[4] GMT: Greenwich Mean Time. Η «ώρα Ελλάδος» είναι δύο ώρες μπροστά (02:00:00)

```
   int tm_hour;    // hour (0 - 23)
   int tm_mday;    // day of month (1 - 31)
   int tm_mon;     // month (0 - 11)
   int tm_year;    // year (Έτος - 1900)
   int tm_wday;    // μέρα εβδομάδας (0 – 6, 0 για Κυριακή)
   int tm_yday;    // μέρα του έτους (0 -365)
   int tm_isdst;   // για την αλλαγή θερινής ώρας.
}; // struct tm
```

Η αντίστροφη λειτουργία γίνεται από τη

```
time_t mktime( tm* t );
```

Πώς μπορούμε να χρησιμοποιήσουμε τα παραπάνω μαζί με τη *Date*; Ας πούμε ότι έχουμε μια τιμή:

```
   Date d1;
```

και θέλουμε να βάλουμε στην *t*1 την τιμή τύπου **time_t** που ισοδυναμεί με *d1*. Δηλώνουμε μια:

```
   tm tm1;
```

και βάζουμε:

```
   tm1.tm_sec = 0;
   tm1.tm_min = 0;
   tm1.tm_hour = 12;   // μεσημέρι
   tm1.tm_mday = d1.dDay;
   tm1.tm_mon = d1.dMonth - 1;
   tm1.tm_year = d1.dYear - 1900;
   tm1.tm_wday = 0;
   tm1.tm_yday = 0;
   tm1.tm_isdst = 0;
```

Μετά χρησιμοποιούμε τη *mktime()*:

```
   t1 = mktime( &tm1 );
```

Αντιστρόφως, αν έχουμε την τιμή της *t1* παίρνουμε την αντίστοιχή της *d1* ως εξής:

```
   tm1 = *localtime( &t1 );
```

και στη συνέχεια:

```
   d1.dYear = tm1.tm_year + 1900;
   d2.dMonth = tm1.tm_month + 1;
   d2.dDay = tm1.tm_day;
```

22.6.1.5 * Υλοποίηση της *forward()*

Με βάση τα παραπάνω υλοποιούμε τη

```
   Date& Date::forward( int dd );
```

ως εξής:

- Δηλώνουμε τις:

```
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
```

- Μετατρέπουμε τις *dd* ημέρες σε 24×60×60×*dd* (= 86400×*dd*) δευτερόλεπτα:

```
   dd *= 86400;      // ημέρες σε δευτερόλεπτα
```

- Προσθέτουμε τη *dd* στην *currT*:

```
   currT += dd;
```

- Τέλος, μετατρέπουμε την τιμή *time_t* που προκύπτει στην «ισοδύναμη» τιμή *Date*:

```
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
```

Να ολόκληρη η μέθοδος:

```
Date& Date::forward( long int dd )
```

```
{
   const unsigned long secsPerDay = 86400;
   tm currD = { 0, 0, 12, dDay, dMonth-1, dYear-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   dd *= secsPerDay;     // ημέρες σε δευτερόλεπτα
   if ( dd < 0 )
   {
      if ( currT < (-dd) )  // currT + dd < 0
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   else // dd >= 0
   {
      if ( dd > LONG_MAX - currT )  // currT + dd > LONG_MAX
         throw DateXptn( "operator+=", DateXptn::outOfLimits,
                         dd/secsPerDay, *this );
   }
   currT += dd;
   currD = *localtime( &currT );
   dYear = currD.tm_year+1900;
   dMonth = currD.tm_mon+1;
   dDay = currD.tm_mday;
   return *this;
} // Date::forward
```

Στην **if** ελέγχουμε κατά πόσον η τιμή που προκύπτει είναι μέσα στα όρια.

Ο τελεστής "**+=**" υλοποιείται με τον ορισμό (**inline**)

```
   Date& operator+=( long int dd ) {  return forward( dd );  }
```

22.6.1.6 * Ο Τελεστής "++" της *Date* (ξανά)

Τώρα, για να είμαστε συνεπείς με αυτά που λέμε στην §22.2, θα πρέπει να ξαναορίσουμε την επιφόρτωση του "++" ως εξής:

```
   Date& operator++() {  return forward( 1 );  }
```

Καλό πράγμα η συνέπεια και η συμβατότητα, αλλά η προηγούμενη μορφή της επιφόρτωσης δεν ήταν ταχύτερη; Ναι, ήταν!

22.6.2 Ο Τελεστής "()" και η Χρήση του

Μια ιδιαίτερη περίπτωση δυαδικού τελεστή με το αντικείμενο αριστερά είναι ο τελεστής κλήσης συνάρτησης "**()**". Με την επιφόρτωση αυτού του τελεστή για μια κλάση *C*:

```
struct C
{
// . . .
   double operator()( double x ) const;
// . . .
};
```

αν έχεις δηλώσει

```
   C f;
```

μπορείς να γράφεις:

```
   if ( f(a)*f(b) < 0 ) m = f(a) + c/2;
// . . .
```

Δηλαδή, χρησιμοποιούμε το αντικείμενο *f* σαν συνάρτηση. Για τον λόγο αυτόν τέτοια αντικείμενα ονομάζονται **συναρτησιακά αντικείμενα** ή **συναρτησοειδή** (function objects, functors).

Όπως καταλαβαίνεις, από το παραπάνω παράδειγμα, αριστερά των παρενθέσεων βρίσκεται το όνομα του αντικειμένου αλλά στα δεξιά –και για την ακρίβεια μέσα στις παρεν-

θέσεις (όπως στην περίπτωση του "`[]`")– πρέπει να υπάρχει λίστα παραμέτρων που στο παράδειγμά μας είναι μονομελής αλλά, γενικώς, μπορεί να είναι πολυμελής ή και κενή.

Πού θα μπορούσε να είναι χρήσιμη αυτή η επιφόρτωση; Η πιο τυπική χρήση: στην αποφυγή της τεχνικής *συναρτήσεων ανάκλησης*. Ας δούμε ένα

Παράδειγμα

Ξαναγυρνούμε στη συνάρτηση *bisection()* στην §14.3. Η χρήση βέλους προς συνάρτηση δεν είναι και τόσο βολική. Να το αλλάξουμε λοιπόν, αλλά πώς;

Στο Παράδ. 2 της §14.7.1 λύσαμε ένα πρόβλημα που είχαμε με «μη βολικές» παραμέτρους καταφεύγοντας σε παραμέτρους περιγράμματος. Μήπως μπορούμε να κάνουμε το ίδιο και εδώ; Δηλαδή: να κάνουμε

- τη *bisection()* περίγραμμα συνάρτησης και
- την *f* της "*f*(*x*) = 0" παράμετρο του περιγράμματος.

Καλό θα ήταν, αλλά *το περίγραμμα δεν δέχεται παράμετρο-συνάρτηση*! Δεχεται όμως παράμετρο κλάση και το πρόβλημά μας λύνεται αν κανονίσουμε να βάζουμε κλάση που τα αντικείμενά της είναι συναρτησοειδή. Γράφουμε λοιπόν:

```
template < class Func >
void bisection( double a, double b, double epsilon,
                double& root, int& errCode )
{
   Func   f;
   double m;

   if ( f(a)*f(b) > 0 )
      errCode = 1;
   else
   {
      while ( fabs(b - a)/2 >= epsilon )
      {
         m = (a + b) / 2;
         if ( f(a)*f(m) <= 0.0 )  b = m;
                             else  a = m;
      } // while
      root = m;
      errCode = 0;
   } // if
} // bisection
```

Όπως βλέπεις, ένα αντικείμενο κλάσης *Func*, όπως το *f*, θα πρέπει να μπορεί να χρησιμοποιηθεί ως συνάρτηση, για να έχουν νόημα τα ""`f(a)`", "`f(b)`", "`f(m)`".

Τώρα, αντί για συναρτήσεις θα γράψουμε κλάσεις:

```
class qF
{
public:
   double operator()( double x ) const
   {  return ( x - log(x) - 2 );  }
}; // qF


class cosF
{
public:
   double operator()( double x ) const
   {  return cos( x );  }
}; // cosF
```

Και να πώς θα γίνει το πρόγραμμα:

```
      bisection< qF >( 0.1, 1.0, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
                     else  cout << " Ρίζα = " << riza << endl;
      bisection< cosF >( 0.0, pi, 1e-5, riza, errCode );
      if ( errCode != 0 )  cout << "Λάθος διάστημα" << endl;
```

```
                    else  cout << " Ρίζα = " << riza << endl;
```

Προσοχή: Ο μεταγλωττιστής θα δημιουργήσει δύο *bisection()* –δύο στιγμιότυπα του περιγράμματος–

- ένα για την εξίσωση $q(x) = 0$ και
- ένα για την εξίσωση $συν(x) = 0$.

Οι κλάσεις των συναρτησοειδών μπορεί να έχουν και αντίστοιχες κλάσεις εξαιρέσεων. Έτσι, μπορούμε να γράψουμε μια *qFXptn* και την *qF*· ακριβέστερα:

```
struct qFXptn
{
   enum { domain };
   char   funcName[100];
   int    errorCode;
   double errDblVal;
   qFXptn( const char* mn, int ec, double dv=0.0 )
      : errorCode( ec ), errDblVal( dv )
   {  strncpy( funcName, mn, 99 );  funcName[99] = '\0';  }
}; // qFXptn

class qF
{
public:
   double operator()( double x ) const
   {
      if ( x <= 0 )
         throw qFXptn( "operator()", qFXptn::domain, x );
      return ( x - log(x) - 2 );
   }
}; // qF
```

Αν στο πρόγραμμά μας βάλουμε:

```
   try {
   // . . .
      bisection< qF >( -1, 1.0, 1e-5, riza, errCode );
   // . . .
   }
   catch( qFXptn& x )
   {
      cout << "domain error (" << x.errDblVal << ")" << endl;
   }
```

θα μας δώσει:

```
domain error (-1)
```

Η συνάρτηση

```
bool mult7( int a )  {  return ( (a%7) == 0 );  }
```

μας επιστρέφει **true** αν το όρισμα είναι διαιρετό δια 7, αλλιώς **false**. Αυτό είναι χαρακτηριστικό παράδειγμα **κατηγορήματος** (predicate).

Δες όμως και το πρόγραμμα

```
#include <iostream>
using namespace std;

struct Mult7
{
   bool operator()( int a )  {  return ( (a%7) == 0 );  }
};

int main()
{
   Mult7 aMult7;

   cout << boolalpha;
```

```
   cout << aMult7(35) << "  " << aMult7(37) << endl;
}
```

που θα δώσει:

```
true  false
```

Και αυτό το συναρτησοειδές, που η συνάρτησή του είναι τύπου **bool**, είναι ένα *κατηγόρημα*.

22.6.2.1 * Μέλος – Περίγραμμα Συνάρτησης

Στην §21.12 είδαμε τα βέλη προς μεθόδους ως εργαλείο για τη χρήση της τεχνικής των συναρτήσεων ανάκλησης. Μήπως μπορούμε να χρησιμοποιήσουμε και στην περίπτωση αυτήν αυτά που μάθαμε παραπάνω για να απαλλαγούμε από τα βέλη προς μεθόδους; Για να κάνουμε κάτι τέτοιο θα πρέπει η *SList*::*toFile()* να γίνει περίγραμμα. Μπορεί να γίνει αυτό; Ναι, *η C++ σου επιτρέπει να βάλεις σε μια κλάση μέλη που να είναι περιγράμματα*. Έτσι στην περίπτωσή μας μπορείς να βάλεις:

```
class SList
{
public:
// . . .
   template < class OutFunc >
   void toFile( ostream& out ) const
   {
      OutFunc toStream;
      for ( ListNode* p(slHead); p != slTail; p = p->lnNext )
         toStream( out, p->lnData );
   } // toFile
// . . .
};
```

Για να το χρησιμοποιήσεις θα πρέπει να γράψεις τρεις κλάσεις, μια για κάθε μια από τις *save()*, *display()* και *writeToTable()* της *GrElmn*:

```
struct SaveGE
{
   void operator()( ostream& bout, const GrElmn& geObj )
   {  geObj.save( bout );  }
}; // SaveGE


struct DisplayGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.display( tout );  }
}; // DisplayGE


struct WriteToTableGE
{
   void operator()( ostream& tout, const GrElmn& geObj )
   {  geObj.writeToTable( tout );  }
}; // writeToTableGE
```

Όταν ο μεταγλωττιστής βρει στο πρόγραμμά μας την

```
      lst.toFile< DisplayGE >( tout );
```

θα εφοδιάσει αυτομάτως την *SList* με την αντίστοιχη μέθοδο.

22.6.3 Αντικείμενο Δεξιά

Εδώ αναφερόμαστε σε επιφορτώσεις της μορφής

```
Trv operator@( Tl a, Tr b )
```

Τέτοιοι τελεστές:

- Αν ο *Tl* είναι κλάση, μπορεί να επιφορτωθούν ως μέθοδοί της, όπως ήδη είδαμε.
- Αν ο *Tr* είναι κλάση, δεν είναι δυνατόν να επιφορτωθούν ως μέθοδοι αφού, όπως έχουμε πει, το αντικείμενο που μας ενδιαφέρει παίζει –στη μέθοδο– ρόλο πρώτης παραμέτρου. Μπορούμε να τους επιφορτώσουμε ως *καθολικές συναρτήσεις*, όπως ακριβώς μάθαμε στην §14.6.4.

Για παράδειγμα, στην §15.5 επιφορτώσαμε τον "`<<`" για τη *Date* ως:

```
ostream& operator<<( ostream& tout, const Date& rhs )
```

Αυτός:

- Θα μπορούσε να επιφορτωθεί ως μέθοδος στην *ostream*, αλλά
- Για τη *Date* θα επιφορτωθεί ως καθολική συνάρτηση.

Η επιφόρτωση που κάναμε στην §19.1.4 είναι μια χαρά:

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

Να τη δηλώσουμε `friend`; Δεν χρειάζεται, αφού οι τρεις "*get*" είναι `inline`.

Παρομοίως γίνεται και η επιφόρτωση του "`<<`" για τη *BString*:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   return tout << rhs.c_str();
} // operator<<( ofstream, BString
```

Αυτός είναι ένας απλός αλλά κάπως επιπόλαιος τρόπος. Η εκτύπωση μέσω της *c_str()* (και γενικώς η χρήση της) τελειώνει στον πρώτο `'\0'` που τυχόν θα βρεθεί. Στην §20.1 επικαλεστήκαμε ως λόγο επιλογής της συγκεκριμένης παράστασης για τα αντικείμενα *BString* το ότι μας επιτρέπει «*να έχουμε και χαρακτήρες* `'\0'` *στην τιμή που αποθηκεύουμε.*» Αν λοιπόν το *bsData* δείχνει κάτι σαν `"ab\0cd"` θα πρέπει η εκτύπωση να τελειώνει στο `'d'` και όχι στο `'b'`. Να πώς μπορεί να γίνει αυτό:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.bsLen; ++k )  tout << rhs.bsData[k];
   return tout;
} // operator<<( ofstream, BString
```

Αν τη γράψεις έτσι θα πρέπει να δηλώσεις και:

```
class BString
{
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
// . . .
```

Αφού όμως και η *length()* και ο `operator[]` είναι μέθοδοι `inline` μπορούμε να γράψουμε χωρίς επιπλέον κόστος:

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < rhs.length(); ++k )  tout << rhs[k];
   return tout;
} // operator<<( ofstream, BString
```

που δεν χρειάζεται να δηλωθεί ως `friend`.

22.7 Αντιμεταθετικοί Τελεστές

Ένας δυαδικός *αντιμεταθετικός τελεστές* είναι ένας τελεστής που δέχεται δύο ορίσματα και έχει την ίδια δράση αν αντιμεταθέσουμε το αριστερό με το δεξιό μέρος· για παράδειγμα ο "`+`", οι τελεστές σύγκρισης κλπ. Στην §22.1 είδαμε με ένα παράδειγμα γιατί δεν μπορούμε

να επιφορτώσουμε τον "==" για τη *BString* με μέθοδο. Παρόμοιο πρόβλημα υπάρχει και με τον "+" και άλλους παρόμοιους τελεστές για την ίδια κλάση (και όχι μόνο).

Οι τελεστές αυτοί υλοποιούνται με *καθολικές συναρτήσεις* που μπορεί να δηλωθούν **friend** για να γίνουν ταχύτερες.

Έτσι, στη *BString* θα κάνουμε την επιφόρτωση του "+" ως

```
   BString operator+( const BString lhs, const BString rhs );
```

Πώς; Αφού έχουμε επιφορτώσει τον "+=" υπάρχει μια πάγια τεχνική.

22.7.1 Από τον "@=" στον "@"

Έχοντας τον "+=" (γενικώς: τον "@=") παίρνουμε τον "+" (γενικώς: τον "@") ως εξής:

```
BString operator+( const BString& lhs, const BString& rhs )
{
   BString fv( lhs );
   fv += rhs;
   return fv;
} // BStribg operator+
```

Έτσι, έχουμε εξασφαλισμένη και τη συμβατότητα με τη λειτουργία του "+=".

Πρόσεξε τον τύπο του αποτελέσματος: δεν είναι *BString*`&` αλλά *BString*. Γιατί; Διότι η επιστρεφόμενη τιμή είναι τοπική μεταβλητή της συνάρτησης και η ζωή της τελειώνει όταν τελειώνει η εκτέλεση της συνάρτησης. Στον "+=" (και την *append*) επιστρεφόμενη τιμή είναι το ίδιο το αντικείμενο (`*this`) που επιζεί και μετά τον τερματισμό εκτέλεσης της μεθόδου.

Πρόσεξε ακόμη ότι η υλοποίηση δεν χρησιμοποιεί μέλη του αντικειμένου και επομένως δεν χρειάζεται να δηλωθεί ως `friend`.

Μπορούμε να χρησιμοποιήσουμε την ίδια τεχνική για να επιφορτώσουμε τον "+" για τη *Date*; Βεβαίως, αλλά πριν το κάνουμε θα πρέπει να σκεφτούμε και κάποια άλλα πράγματα.

Κατ' αρχήν πρέπει να συμφωνήσουμε στο νόημα της πράξης έτσι ώστε να είναι συμβατή με τον "+=". Τι σημαίνει συμβατή; Σημαίνει ότι οι: "`d = d + 5`" και "`d += 5`" ("`d.forwatd(5)`") θα έχουν το ίδιο αποτέλεσμα. Αυτό μας λέει ότι μπορούμε να χρησιμοποιήσουμε την παραπάνω τεχνική.

Το άλλο σημείο που πρέπει να προσέξουμε είναι η αντιμεταθετικότητα: μετά τη δήλωση:

```
   Date d( 2007, 10, 15 ), d1;
```

οι:

```
   d1 = d + 5;
```

και

```
   d1 = 5 + d;
```

θα πρέπει να έχουν το ίδιο αποτέλεσμα.

Δηλαδή, έχουμε να υλοποιήσουμε δύο τελεστές με το ίδιο σύμβολο:[5]

`+`: *Date* × `int` ↛ *Date*

`+`: `int` × *Date* ↛ *Date*

Πέρα από την αντιμεταθετικότητα του τελεστή έχουμε και μια *ασύμμετρία* λόγω της διαφοράς τύπου των δύο ορισμάτων. Έτσι, η επιφόρτωση του γίνεται όπως και στη *BString* αλλά είναι διπλή αφού έχουμε δύο συναρτήσεις:

```
Date operator+( const Date& lhs, int rhs )
{
   Date fv( lhs );
   fv += rhs;
   return fv;
} // Date operator+
```

[5] Γιατί μερικές συναρτήσεις; Δες την υλοποίηση του "+=" και θα καταλάβεις.

```
Date operator+( int lhs, const Date& rhs )
{
   return ( rhs + lhs );
} // Date operator+
```

Και εδώ, ο τύπος του αποτελέσματος δεν είναι *Date*`&` αλλά *Date*, για ίδιους λόγους που είπαμε και στη *BString*.

22.7.2 Σύγκριση Ημερομηνιών

Μια άλλη κατηγορία «συμμετρικών»[6] τελεστών που υλοποιούνται με καθολικές συναρτησεις (με δύο παραμέτρους) είναι οι τελεστές σύγκρισης. Έτσι, οι επιφορτώσεις των "`==`" και "`<`" για την κλάση *Date*, όπως τις κάναμε στην §19.1.4 είναι μια χαρά.

Τώρα θέλουμε να τονίσουμε μια καλή προγραμματιστική πρακτική που μπορεί να σε προφυλάξει από «δύσκολα» προγραμματιστικά λάθη:[7]

- ***Όταν επιφορτώνεις έναν τελεστή σύγκρισης "`@`" επιφόρτωσε και τον αντίθετό του με χρήση του "`@`".***

Αυτό σημαίνει ότι μετά τους "`==`" και "`<`" θα πρέπει να επιφορτώσουμε και τους "`!=`" και "`>=`" και μάλιστα ως εξής:

```
bool operator!=( const Date& lhs, const Date& rhs )
{  return !(lhs == rhs);  }

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

Παρατήρηση: ▶

Αν δηλώσεις τον δημιουργό της *Date* ως (§21.13.1):

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

δεν μπορείς να γράψεις "`d == 2013`" ή "`2013 == d`" αλλά, υποχρεωτικώς, "`d == Date(2013)`" ή "`Date(2013) == d`" αντιστοίχως. Έτσι, δεν υπάρχει πια πρόβλημα αν επιφορτώσουμε τον "==" (και τους άλλους τελεστές σύγκρισης) με μεθόδους (με μια παράμετρο), π.χ.:

```
bool Date::operator==( const Date& rhs ) { /*. . .*/ }
```

Για τη *BString*, στον δημιουργό της οποίας δεν βάλαμε το "`explicit`", αυτό δεν ισχύει και η καθολική συνάρτηση είναι η μόνη σωστή επιλογή.◀

22.7.3 Οι Τελεστές Σύγκρισης της *BString*

Στη συνέχεια θέλουμε να προχωρήσουμε στην επιφόρτωση των τελεστών σύγκρισης ("`==`", "`<`", κλπ) στον τύπο *BString*. Θα πρέπει όμως πρώτα να συζητήσουμε τις συγκρίσεις ορμαθών χαρακτήρων, που θα χρειαστούμε στη συνέχεια.

Ας πούμε ότι έχουμε δύο πίνακες:

```
   char lhs[10], s2[12];
```

οι οποίοι περιέχουν κείμενα με μήκη *len1* και *len2*. Βάζουμε:

```
   minLen = min( len1, len2 );
```

---

[6] Γιατί «*συμμετρικών*» και όχι «*αντιμεταθετικών*»; Φυσικά, ο "`<`" δεν είναι αντιμεταθετικός: αν η *lhs* `<` *rhs* έχει τιμή `true` τότε η *rhs* `<` *lhs* έχει τιμή `false`. Εδώ με το *συμμετρικός* εννοούμε οτι από *συντακτική άποψη* ό,τι μπορεί να μπει στο ένα μέρος μπορεί να μπει και στο άλλο.

[7] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposites (such as "==" and "!="), it is appropriate to define both.*» Την πρωτοείδαμε στο Project 2 (§Prj02.3).

Αφού υπενθυμίσουμε ότι στις θέσεις *lhs*[*len*1] και *s*2[*len*2] υπάρχει ο φρουρός '\0', θεωρούμε τις:

```
k = 0;
while ( lhs[k] == s2[k] && k < minLen-1 ) ++k;
d = static_cast<int>(lhs[k]) - static_cast<int>(s2[k]);
```

Ας δούμε την τιμή που θα πάρει η *d* με μερικά παραδείγματα:

α) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2 τα ίδια: abcd. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η **while** θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == *s*2[3] == '\0' η *d* θα πάρει τιμή 0.

β) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: abce. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η while θα σταματήσει και πάλι όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'e' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('e') = 100 – 101 = –1 < 0.

γ) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: abcc. Θα έχουμε *minLen* = *min*(4, 4) = 4 και αφού *lhs*[*k*] == *s*2[*k*] για κάθε *k* από 0 μέχρι 2 η while θα σταματήσει όταν *k* == 3. Επειδή όμως *lhs*[3] == 'd' και *s*2[3] == 'c' η *d* θα πάρει τιμή static_cast<int>('d') – static_cast<int>('c') = 100 – 99 = 1 > 0.

Αυτή είναι η βασική ιδέα:

- Αν το *lhs προηγείται αλφαβητικώς* (λεξικογραφικώς) *του s*2 (περίπτωση β) η *d* παίρνει *αρνητική* τιμή.
- Αν το *lhs έπεται αλφαβητικώς του s*2 (περίπτωση γ) η *d* παίρνει *θετική* τιμή.
- Αν το *lhs είναι ίσο με το s*2 (περίπτωση α) η *d* παίρνει τιμή 0 (μηδέν).

Η σύγκριση γίνεται, κατ' αρχήν, με βάση τη θέση των χαρακτήρων στον πίνακα χαρακτήρων του υπολογιστή. Το «κατ' αρχήν» θα το συζητήσουμε αργότερα.

Δες μερικά παραδείγματα ακόμη:

δ) Έστω ότι στο *lhs* έχουμε ab και στο *s*2: abcd. Θα έχουμε *minLen* = *min*(2, 4) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

ε) Έστω ότι στο *lhs* έχουμε abcd και στο *s*2: ab. Θα έχουμε *minLen* = *min*(4, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[1] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

στ) Έστω ότι στο *lhs* έχουμε ab␣ και στο *s*2: ab. Θα έχουμε *minLen* = *min*(3, 2) = 2 και αφού *lhs*[*k*] == *s*2[*k*] για *k* == 0 η while θα σταματήσει όταν *k* == 1. Επειδή όμως *lhs*[1] == 'b' και *s*2[3] == 'b' η *d* θα πάρει τιμή static_cast< int>('b') – static_cast<int>('b') = 98 – 98 = 0.

Εδώ έχουμε τρεις περιπτώσεις που παίρνουμε *d* == 0 χωρίς να έχουμε ισότητα. Τι σημαίνει όμως αυτό; Ότι όσο μπορούμε να συγκρίνουμε έχουμε ισότητες χαρακτήρων. Άρα, ο ορμαθός που τελειώνει μέχρι εκεί που μπορούμε να συγκρίνουμε προηγείται ενώ ο άλλος, με το μεγαλύτερο μήκος έπεται.

Με βάση τα παραπάνω, γράφουμε μια συνάρτηση-εργαλείο, εσωτερική της *BString*, για την υλοποίηση των μεθόδων που θα προδιαγράψουμε στη συνέχεια[8]:

```
int BString::stringCmpr( const char* lhs, const char* rhs, int n )
{
   int k( 0 );
   while ( lhs[k] == rhs[k] && k < n-1 ) ++k;
   return ( static_cast<int>(lhs[k]) - static_cast<int>(rhs[k]) );
```

[8] Η υλοποίηση που θα δώσουμε είναι μια παραλλαγή της υλοποίησης που δίνεται στην STL της Silicon Graphics Inc (SGI).

```
} // BString::stringCmpr
```

Η *stringCmpp()*[9] συγκρίνει τους πρώτους *n* (0 .. *n*–1) χαρακτήρες των *lhs* και *s*2 και επιστρέφει:

- *αρνητική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs προηγούνται αλφαβητικώς* (λεξικογραφικώς) των *n* πρώτων χαρακτήρων *του s*2,
- *θετική* τιμή αν οι *n* πρώτοι χαρακτήρες του *lhs έπονται αλφαβητικώς* των *n* πρώτων χαρακτήρων *του s*2,
- 0 (μηδέν) αν οι *n* πρώτοι χαρακτήρες του *lhs είναι ίσοι* με τους αντίστοιχους *n* πρώτους χαρακτήρες *του s*2.

Πρόσεξε ότι δεν ελέγχουμε πουθενά την τιμή της *n*: αυτό θα πρέπει να το κάνει η κάθε μέθοδος που καλεί τη *stringCmpr()*.

Η *string* έχει τις εξής μεθόδους[10] για σύγκριση:

```
   int compare( const BString& rhs ) const;
   int compare( int pos, int n, const BString& rhs) const;
   int compare( int pos, int n,
                const BString& rhs, int pos2, int n2 ) const;
```

Όλες οι μορφές της *compare()* επιστρέφουν μιαν ακέραιη τιμή με την ίδια λογική που έχουμε για τη *stringCmpr()*. Με τη βοήθεια της *stringCmpr()* θα υλοποιήσουμε την πρώτη από αυτές τις μεθόδους για τη *BString*. Για τα παρακάτω παραδείγματα υποθέτουμε ότι έχουμε δηλώσει:

```
   BString bs1, bs2;
```

Η παράσταση **bs1.compare(bs2)** θα πρέπει να επιστρέφει τιμή *μικρότερη από, ίση με* ή *μεγαλύτερη από* 0 αν, αντιστοίχως, η τιμή της *bs*1 *προηγείται από, είναι ίση με* ή *έπεται της* τιμής της *bs*2. Μπορούμε λοιπόν να ορίσουμε:

```
int BString::compare( const BString& rhs ) const
{
   int fv( stringCmpr(bsData, rhs.bsData, min(bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = bsLen - rhs.bsLen;
   return fv;
} // BString::compare
```

Ας έλθουμε τώρα στους τελεστές. Θα επιφορτώσουμε τους τελεστές σύγκρισης με *καθολικές συναρτήσεις*. Μπορούμε να γράψουμε:

```
bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::stringCmpr(lhs.bsData, rhs.bsData,
                               min(lhs.bsLen, rhs.bsLen)) );
   if ( fv == 0 )  fv = lhs.bsLen - rhs.bsLen;
   return ( fv == 0 );
} // operator==
```

Φυσικά, όπως και στην περίπτωση του **operator<** της *Date*, θα πρέπει να δηλώσουμε μέσα στην κλάση ως **friend** τη συνάρτηση (τελεστή) **operator==**:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
public:
   BString( const char* cs = "" );
// ...
```

και ο ορισμός του τελεστή που δώσαμε πιο πάνω γίνεται νόμιμος.

---

[9] Η *stringCmpr()* είναι μια βοηθητική συνάρτηση, σαν τις *lastDay()* και *leapYear()* της *Date*.

[10] Δεν είναι ακριβώς έτσι. Αν ψάξεις λίγο το αρχείο string θα δεις ότι είναι πιο πολύπλοκα.

22.7.3.1 * Η Σειρά Ταξινόμησης

Οι συγκρίσεις που είδαμε παραπάνω στηρίζονται στη θέση των χαρακτήρων μέσα στο σύνολο χαρακτήρων του υπολογιστή. Αυτό δεν μας βολεύει καθόλου όταν θέλουμε να ταξινομήσουμε λέξεις κατά τη συνήθη αλφαβητική σειρά. Όπως είναι τοποθετημένα τα ελληνικά γράμματα στο πρότυπο ΕΛΟΤ 928 θα έχεις τα εξής περίεργα: το όνομα `ΝΙΚΟΛΑΪΔΗΣ` ταξινομείται μετά το `ΝΙΚΟΛΑΚΗΣ` και η λέξη `μικροψυχία` πριν από τα `μικροϋλικά`.

Αυτό το πρόβλημα λύνεται με τη **σειρά ταξινόμησης** (collating sequence). Αυτή είναι μια συνάρτηση που καθορίζει τη σειρά των χαρακτήρων για την ταξινόμηση. Για σύνολο 256 χαρακτήρων μπορείς να την υλοποιήσεις πολύ εύκολα με έναν πίνακα:

```
   int cs[256];
```

Σε κάθε στοιχείο του cs βάζουμε ως τιμή τη σειρά με την οποία θέλουμε να ταξινομείται. Π.χ.:

```
   for ( int k = 0; k <= 255; ++k )  cs[k] = k;
// ...
// Λατινικοί
   cs['A'] = cs['a'] = 65;
   cs['B'] = cs['b'] = 66;
// ...
// Ελληνικοί
   cs[256+'Α'] = cs[256+'Ά'] = cs[256+'α'] = cs[256+'ά'] = 161;
   cs[256+'Β'] = cs[256+'β'] = 162;
// ...
   cs[256+'Υ'] = cs[256+'Ύ'] = cs[256+'Ϋ'] = cs[256+'υ']
               = cs[256+'ύ'] = cs[256+'ϋ'] = cs[256+'ΰ'] = 213;
// ...
```

Θυμίσου ότι τα ελληνικά γράμματα, ως σταθερές τύπου (`signed`) `char`, έχουν αρνητική τιμή[11]. Αν δεν καταλαβαίνεις τι ακριβώς κάνουμε γύρισε πίσω στο Κεφ. 3.

Αφού η βάση του μηχανισμού σύγκρισης είναι η *stringCmpr()* την αλλάζουμε ως εξής:

```
int BString::stringCmprCS( const char* lhs, const char* rhs, int n ) const
{
   int k = 0;

   while ( cs[ lhs[k] ] == cs[ rhs[k] ] && k < n-1) ++k;
   return ( cs[ lhs[k] ] - cs[ rhs[k] ] );
} // BString::stringCmprCS
```

22.8 Τελεστές για τη *Vector3*

Επιστρέφουμε τώρα στην κλάση *Vector3*, που είδαμε στο Project 2, για να ξαναδούμε –όπως έχουμε υποσχεθεί– τις επιφορτώσεις τελεστών.

Εξ αρχής να πούμε το εξής: Η *Vector3* έχει τετριμμένη αναλλοίωτη, όλα τα μέλη είναι ανοικτά και έτσι δεν υπάρχει πρόβλημα για την επιφόρτωση όλων των τελεστών με καθολικές συναρτήσεις. *Αυτά που δώσαμε στο Project 2 είναι απολύτως σωστά*. Αυτά που θα πούμε εδώ είναι παραδείγματα εφαρμογής κάποιων από τις συνταγές που είδαμε στο παρόν κεφάλαιο.

Ας ξεκινήσουμε με τον (προθεματικό) *ενικό τελεστή* "`-`" (πρόσημο). Αν **v** = (*x*, *y*, *z*) τότε `-`**v** = (`-`*x*, `-`*y*, `-`*z*). Σύμφωνα με τη συνταγή της §14.6.4 θα έχουμε:

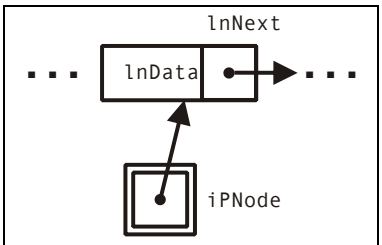
```
   Vector3 operator-( Vector3 rhs )
```

(ο τελεστής δεν αλλάζει το αντικείμενο) και σύμφωνα με τη συνταγή της §22.3.1 θα πρέπει να επιφορτώσουμε με μέθοδο:

```
   Vector3 operator-() const {  return Vector3( -x, -y, -z );  }
```

[11] Κανονικά πρέπει να γράψουμε `cs[256+static_cast<int>('A')] = 193` αλλά και έτσι περνάει.

Με μεθόδους θα πρέπει να επιφορτώσουμε και τους μη-αντιμεταθετικούς τελεστές με το αντικείμενο αριστερά "`+=`", "`-=`", "`*=`":



Σχ. 22-1 Ένας προσεγγιστής: το μοναδικό του μέλος, *iPNode*, είναι ένα βέλος που δείχνει κάποιον κόμβο της λίστας.

```
Vector3& Vector3::operator+=( const Vector3& rhs )
{
   x += rhs.x;  y += rhs.y;  z += rhs.z;
   return *this;
} // Vector3::operator+=

Vector3& Vector3::operator-=( const Vector3& rhs )
{
   x -= rhs.x;  y -= rhs.y;  z -= rhs.z;
   return *this;
} // Vector3::operator-=

Vector3& Vector3::operator*=( double rhs )
{
   x *= rhs;  y *= rhs;  z *= rhs;
   return *this;
} // operator*=( const Vector3&
```

Μέθοδοι θα γίνουν και οι *abs()* και *abs2()*:

```
   double abs2() const {  return x*x + y*y + z*z;  }
   double abs() const {  return sqrt( abs2() );  }
```

Οι υπόλοιπες καθολικές συναρτήσεις παραμένουν αλλά αλλάζουν οι ορισμοί των "`operator+`", "`operator-`", "`operator*`" που υλοποιούνται όπως μάθαμε στην §22.6.1, π.χ.:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv( lhs );
   fv += rhs;
   return fv;
} // operator+( const Vector3&
```

22.9 Διασχίζοντας τη Λίστα με τον "++" – Προσεγγιστές

Στην §21.11 λέγαμε: «*Στις διάφορες εφαρμογές που χρησιμοποιούμε λίστες χρειάζεται να διασχίσουμε μια λίστα για πολλές και πολύ διαφορετικές επεξεργασίες του περιεχομένου. ... η διάσχιση της λίστας δεν μπορεί να "κρυφτεί" σε κάποια μέθοδο.*» Έτσι, δίνουμε στο πρόγραμμα-πελάτη τη δυνατότητα να δίνει την εντολή "`p = p->lnNext`" με τον κίνδυνο να πάρει η *p->lnNext* τιμή που βγάζει από τη λίστα.

Τώρα ας εξετάσουμε την εξής ιδέα:

- όπως η "`++n`" προχωρεί την τιμή της *n* στον επόμενο ακέραιο,
- όπως η "`++d`" προχωρεί την τιμή της *d* στην επόμενη ημερομηνία

να επιφορτώσουμε τον "`++`" ώστε να προχωρεί τον *p* στον επόμενο κόμβο της λίστας. Και ακόμη να επιφορτώσουμε τον (ενικό) "`*`" ώστε η "`*p`" να μας δίνει το *p->lnData*. Έτσι δεν θα υπάρχει λόγος να «ανοίγουμε» τη δομή της λίστας στο πρόγραμμα-πελάτη.

Καλή ιδέα, αλλά έχει ένα προβληματάκι: η C++ δεν μας επιτρέπει να επιφορτώνουμε τελεστές για τους πρωτογενείς τύπους. Όμως αυτό το πρόβλημα λύνεται: Γράφουμε μια κλάση **προσεγγιστή** (iterator) που έχει ένα μόνον μέλος, το βέλος μας! Στο Σχ. 22-1 βλέπεις τι περίπου εννοούμε. Το βλέπεις και στη συνέχεια:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
```

```
public:
   class Iterator
   {
   public:
// . . .
      Iterator& operator++()
      {  iPNode = iPNode->lnNext;  return *this;  };
      GrElmn& operator*() {  return ( iPNode->lnData );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
// . . .
private:
   ListNode* slHead;
   ListNode* slTail;
// . . .
}; // SList
```

Εδώ όμως βλέπεις και άλλα:

- Η κλάση *SList* έχει τώρα δύο περιοχές **private**.
- Στην πρώτη περιοχή **private** έχουμε «κρύψει» τον ορισμό της *ListNode* που πρέπει να προηγείται της δήλωσης

```
      ListNode* iPNode;
```

- Στη νέα κλάση, που είναι κλάση «περιτυλίγματος», βλέπεις την επιφόρτωση των "`++`" και "`*`". Εδώ βρίσκεται το ενδιαφέρον!

Τι άλλο πρέπει να αλλάξουμε στην *SList*; Προφανώς τις *begin()* και *end()*· δεν είναι δυνατόν πια να επιστρέφουν βέλος. Θα τις κάνουμε:

```
   Iterator begin() {  return Iterator( slHead );  }
   Iterator end() {  return Iterator( slTail );  }
```

Προφανώς χρειάζεται να γράψουμε και έναν δημιουργό για τη νέα κλάση:

```
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
```

Πώς θα διασχίσουμε τη λίστα για να φυλάξουμε το περιεχόμενό της; Έτσι:

```
      for ( SList::Iterator it(lst.begin());
            it != lst.end();
            ++it )
         writeRandom( *it, bInOut );
```

Εδώ τα βλέπεις όλα:

- Χρησιμοποιούμε τον προσεγγιστή *it* για να πάμε από `lst.begin()` μέχρι `lst.end()`.
- Προχωρούμε με "`++it`".[12]
- Σε κάθε βήμα φυλάγουμε το περιεχόμενο ενός κόμβου που δίνεται από την αποπαραπομπή "`*it`".

Βλέπεις όμως ότι μας λείπει και κάτι: δεν επιφορτώσαμε τον "`!=`". Μπορούμε να τον επιφορτώσουμε με καθολική συνάρτηση:

```
bool operator!=( const SList::Iterator& a, const SList::Iterator& b )
{  return ( a.iPNode != b.iPNode );  }
```

αλλά θα πρέπει να την δηλώσουμε ως "`friend`".

Αφού έχουμε δηλώσει "`explicit Iterator`" δεν υπάρχει πρόβλημα αν κάνουμε την επιφόρτωση με μέθοδο:

---

[12] Οι προσεγγιστές που βλέπουμε εδώ έχουν μια διεύθυνση κίνησης: από την αρχή της λίστας προς το τέλος της («κινούνται» μόνον με τον "`++`"). Ένας τέτοιος προσεγγιστής καλείται **πρόσθιος** (forward) **προσεγγιστής**. Αν κάθε κόμβος είχε και βέλος προς τον προηγούμενο κόμβο θα ήταν δυνατή κίνηση του προσεγγιστή και από το τέλος προς την αρχή (και επιφόρτωση του "`--`"). Ένας τέτοιος προσεγγιστής λέγεται **αμφίδρομος** (bidirectional).

```
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
```

Κατά τη συνήθειά μας θα ορίσουμε και την:

```
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
```

Κλάση εξαιρέσεων θα γράψουμε; Ναι –κατ' αρχήν– διότι οι δύο βασικές επιφορτώσεις είναι ανοικτές σε λάθος χρήση. Δεν είναι όμως και λάθος να χρησιμοποιήσουμε την *SListXptn*.

- Ο "++" δεν μπορεί να χρησιμοποιηθεί αν έχουμε φτάσει στον φρουρό:

```
SList::Iterator& SList::Iterator::operator++()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator++", SListXptn::listEnd );
   iPNode = iPNode->lnNext;
   return *this;
} // SList::Iterator::operator++
```

- Το ίδιο ισχύει και για τον "*":

```
GrElmn& SList::Iterator::operator*()
{
   if ( iPNode->lnNext == 0 )
      throw SListXptn( "Iterator::operator*", SListXptn::listEnd );
   return ( iPNode->lnData );
} // SList::Iterator::operator*
```

Ξαναδίνουμε το τμήμα του ορισμού της *SList* που έχει τις αλλαγές:

```
class SList
{
private:
   struct ListNode
   {
      GrElmn    lnData;
      ListNode* lnNext;
   }; // ListNode
public:
   class Iterator
   {
   friend bool operator!=( const Iterator& a, const Iterator& b);
   public:
      explicit Iterator( ListNode* p = 0 )  {  iPNode = p;  }
      Iterator& operator++();
      GrElmn& operator*();
      bool operator!=( const Iterator& rhs ) const
      {  return ( iPNode != rhs.iPNode );  }
      bool operator==( const Iterator& rhs ) const
      {  return ( !(*this != rhs) );  }
   private:
      ListNode* iPNode;
   }; // Iterator

   SList();
   SList( const SList& rhs );
   ~SList();
   SList& operator=( const SList& rhs );
   void swap( SList& rhs );
   Iterator begin()  {  return Iterator( slHead );  }
   Iterator end()  {  return Iterator( slTail );  }
// . . .
}; // SList

struct SListXptn
{
   enum { noMemory, notFound, listEnd };
// . . .
```

```
}; // SListXptn
```

Παρατήρηση:▶

Τώρα που έχουμε τον προσεγγιστή και επιφορτώσαμε τους "`++`" και "`*`" –και έχοντας επιφορτώσει τον "`==`" για τη *GrElmn*– μπορούμε να χρησιμοποιήσουμε την (*std*::)*find()* για αναζήτηση στη λίστα.

Για να τη δοκιμάσεις βάλε στο πρόγραμμα της §21.11 τη νέα μορφή της *SList*, μην ξεχάσεις την "`include <algorithm>`" και στη **main**, μετά το τέλος της **do-while**, βάλε άλλη μια:

```
      do {
         readAtNo( maxAtNo, aa );
         if ( aa != 0 )
         {
            SList::Iterator it;
            it = std::find( lst.begin(), lst.end(), GrElmn(aa) );
            if ( it != lst.end() )  (*it).display( cout );
                                else  cout << "not found" << endl;
         }
      } while ( aa != 0 );
```

Σημείωσε τους ατομικούς αριθμούς των στοιχείων που θα βάλεις στη λίστα με την πρώτη **do-while** και σύγκρινε με αυτά που θα σου βγάζει η δεύτερη.◀

22.9.1 Επιφόρτωση του "->"

Αν επιφορτώσουμε τον "`->`" για την κλάση *SList*::*Iterator* μπορούμε αντί για

"`(*it).display(cout)`"

να γράψουμε

"`it->display(cout)`"

Ας δούμε πώς γίνεται αυτή η επιφόρτωση.

Ο "`->`" είναι, κατ' αρχήν, ένας δυαδικός τελεστής: αριστερά υπάρχει (για την περίπτωσή μας) ένας προσεγγιστής και δεξιά ένα μέλος ή μια μέθοδος της κλάσης που περιέχει η λίστα. Στη συνάρτηση που θα γράψουμε δεν υπάρχει δεύτερη παράμετρος (αντίστοιχη του δεξιού μέρους). Η συνάρτηση επιστρέφει τη διεύθυνση αυτού (βέλος προς αυτό) που επιστρέφει η `operator*()`. Και αφού η `operator*()` επιστρέφει "`iPNode->lnData`" θα έχουμε:

```
      GrElmn* operator->() const
      {
         if ( iPNode->lnNext == 0 )
            throw SListXptn( "Iterator::operator->",
                             SListXptn::listEnd );
         return ( &(iPNode->lnData) );
      } // SList::Iterator::operator->
```

Η επιστρεφόμενη τιμή –που είναι βέλος– αντικαθιστά στην παράσταση τον προσεγγιστή και ο "`->`" δρα σε αυτό το βέλος. Στην περίπτωσή μας θα έχουμε:

`(&(iPNode->lnData))->display(cout)`

ή αλλιώς:

`(iPNode->lnData).display(cout)`

Αντί για "`it->display(cout)`" μπορείς να γράψεις:

"`(it.operator->())->display(cout)`"

22.10 * Απόκρυψη Υλοποίησης – Τεχνική "pimpl"

Και τώρα που τα μάθαμε όλα ας κλείσουμε μια εκκρεμότητα που έχουμε από την §19.3: να δούμε πώς αποκρύπτουμε την υλοποίηση της κλάσης (για κάποιον από τους λόγους που είδαμε στην §19.3.1).

Αυτό που θα θέλαμε να κάνουμε είναι το εξής: Να παραδίδουμε στον προγραμματιστή που θα χρησιμοποιήσει μια κλάση *C* (που έχουμε γράψει) το αρχείο **C.obj** (ή **C.o**) και το αρχείο **C.h** με τον ορισμό της κλάσης στον οποίον όμως θα υπάρχει μόνον το μέρος "**public**" με τις δηλώσεις των μεθόδων. Το μέρος "**private**" και οι ορισμοί των μεθόδων θα πρέπει να βρίσκονται μεταγλωττισμένα στο **C.obj**.

Ο συνδέτης δεν θα μας επιτρέψει κάτι τέτοιο. Θα μας επιτρέψει όμως κάτι που μοιάζει με αυτό: ένα μέρος "**private**" που θα έχει μέσα μόνο ένα βέλος. Ας δούμε αυτή τη λύση με ένα συγκεκριμένο παράδειγμα.

Η κλάση *BString*, όπως την έχουμε αναπτύξει μέχρι τώρα, είναι:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs ) { return (*this = rhs); }
   const char* c_str() const;
   size_type length() const {  return bsLen;  }
   bool empty() const {  return ( bsLen == 0 );  }
   char& at( int k ) const;
   char& operator[]( int pos ) const {  return bsData[pos];  }
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs )
   { return (*this += rhs); }
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τώρα θα κάνουμε την εξής αλλαγή: Θα ορίσουμε μια άλλη κλάση:

```
struct BStringImpl
{
   enum { bsIncr = 16 };
   char*     bsData;
   size_type bsLen;
   size_type bsReserved;

   static size_type cStrLen( const char* cs );
   static int stringCmpr( const char* lhs, const char* rhs,
                          int n );
}; // BStringImpl
```

και στο μέρος "**private**" της κλάσης θα βάλουμε μόνο ένα βέλος:

```
private:
   BStringImpl* bsHandle;
}; // BString
```

Για να μπορέσουμε να κάνουμε αυτήν τη δήλωση για το *bsHandle* θα πρέπει ο μεταγλωττιστής να βρει *προειδοποιητική δήλωση* της *BStringImpl*:

```
   struct BStringImpl;
```

που λέει ότι ακολουθεί ορισμός αυτής της κλάσης.

Μπορούμε να βάλουμε αυτήν τη δήλωση πριν από τον ορισμό της *BString* (στο `BString.h`). Είναι όμως προτιμότερο να τη βάλουμε μέσα στον ορισμό της κλάσης. Θα δεις στη συνέχεια γιατί αυτό μας συμφέρει.

Στο μέρος "`public`" δεν θα αφήσουμε οτιδήποτε έχει σχέση με υλοποίηση:

```
class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );
```

Τα υπόλοιπα θα πάνε στο `BString.cpp`. Εκεί θα πάνε και ο ορισμοί της νέας κλάσης και των μεθόδων της.

Αν θέλεις μπορείς, όπως κάνουμε μέχρι τώρα, να βάλεις όσα αφορούν την κλάση *BStringImpl* σε χωριστά αρχεία `BStringImpl.h` και `BStringImpl.cpp`. Στην περίπτωση αυτή θα πρέπει να βάλεις "`#include "BStringImpl.h"`" και "`#include "BStringImpl.cpp"`" στην αρχή του `BString.cpp`.

Σημείωση: ▶
Τώρα μπορείς να καταλάβεις και το (συνηθισμένο) όνομα της τεχικής: ***p***ointer to (ή ***p***rivate) ***impl***ementation (βέλος προς την υλοποίηση). Θα συναντήσεις ακόμη τα ονόματα "handle class", "Cheshire Cat" και "compiler firewall". Η *BStringImpl* είναι μια **αδιαφανής** (opaque) δομή δεδομένων ενώ **αδιαφανές** ονομάζεται και το βέλος *bsHandle*· ονομάζεται ακόμη και **λαβή** (handle).

Λαβές για αδιαφανείς δομές δεδομένων χρησιμοποιούνται σε μικρή ή μεγάλη έκταση στις API για διάφορα προγραμματιστικά περιβάλλοντα.◀

Ας δούμε τώρα μερικά βασικά σημεία της υλοποίησης που θα αλλάξουν.

Ο ερήμην δημιουργός θα είναι:

```
BString::BString( const char* rhs )
{
   try {  bsHandle = new BStringImpl( rhs );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Αυτός ο δημιουργός μας επιβάλλει να γράψουμε έναν δημιουργό:

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

για την *BStringImpl*:

```
BString::BStringImpl::BStringImpl( const char* rhs )
{
   bsLen = cStrLen( rhs );
```

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

   bsData = new char[bsReserved];
   for ( int k(0); k < bsLen; ++k )  bsData[k] = rhs[k];
} // BString::BStringImpl::BStringImpl
```

Πρόσεξε ότι η νέα κλάση αναφέρεται ως "`BString::BStringImpl`".

Αν αποτύχει η "`new`" και ριχτεί *bad_alloc* θα συλληφθεί από την "`catch`" του δημιουργού της *BString*. Και αν σε κάποιο πρόγραμμα έχουμε τη δήλωση:

```
   BStringImpl a;
```

και κατά την εκτέλεση του ερήμην δημιουργού ριχτεί μια *bad_alloc* πού θα συλληφθεί; Το ότι η κλάση μας είναι δηλωμένη στο μέρος "`private`" της *BString* μας εξασφαλίζει ότι δεν θα υπάρξει τέτοια δήλωση· θα την «απαγορεύσει» ο μεταγλωττιστής.

Αφού κάθε αντικείμενο δεσμεύει δυναμική μνήμη –για ένα αντικείμενο *BStringImpl*– θα πρέπει να γράψουμε δημιουργό αντιγραφής:

```
BString::BString( const BString& rhs )
{
   try {  bsHandle = new BStringImpl( *(rhs.bsHandle) );  }
   catch( bad_alloc& )
   {  throw BStringXptn( "BString", BStringXptn::allocFailed );  }
} // BString::BString
```

Στη "`new BStringImpl( *(rhs.bsHandle) )`" καλείται ο δημιουργός αντιγραφής της *BStringImpl*. Αφού ένα αντικείμενο αυτής της κλάσης δεσμεύει επίσης μνήμη για το *bsData* θα πρέπει να γράψουμε δημιουργό αντιγραφής και για αυτήν:

```
BString::BStringImpl::BStringImpl( const BString::BStringImpl& rhs )
{
   bsData = new char[rhs.bsReserved];
   bsReserved = rhs.bsReserved;
   for ( int k(0); k < rhs.bsLen; ++k )
      bsData[k] = rhs.bsData[k];
   bsLen = rhs.bsLen;
} // BString::BStringImpl::BStringImpl
```

Αυτά που είπαμε για τις *bad_alloc* από τους ερήμην δημιουργούς των δύο κλάσεων ισχύουν και για τους δημιουργούς αντιγραφής.

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε (επιφορτώσουμε) και τελεστή εκχώρησης:

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

Αυτός είναι ολόιδιος με αυτόν που γράψαμε στην §21.6.1 (τελευταία μορφή) αλλά έχουμε διαφορετική *swap()*:

```
void BString::swap( BString& rhs )
{
   std::swap( bsHandle, rhs.bsHandle );
} // BString::swap
```

Σύμφωνα με τον *κανόνα των τριών* θα πρέπει να γράψουμε τελεστή εκχώρησης και για τη *BStringImpl*:

```
BString::BStringImpl& BString::BStringImpl::operator=(
                                          const BString::BStringImpl& rhs )
```

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

Βέβαια, προς το παρόν μας είναι άχρηστος αλλά μπορεί να μας χρειαστεί αν συνεχίσουμε την ανάπτυξη της *BString*.

Οι καταστροφείς είναι:

```
BString::~BString() {  delete bsHandle;  }
```

και

```
BString::BStringImpl::~BStringImpl() {  delete bsData;  }
```

Άλλες μεθόδους για τη *BStringImpl* θα γράψουμε; Μόνον τη *swap()*:

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

Όπως βλέπεις πρόκειται για την παλιά `BString::swap` σε νέα συσκευασία.

Οι μέθοδοι της *BString* πρέπει να ξαναγραφούν με τις εξής τροποποιήσεις: Θα πρέπει να γράφεις

- το πρόθεμα "`bsHandle->`" πριν από τα *bsData, bsLen, bsReserved*.
- το πρόθεμα "`BStringImpl::`" πριν από τη σταθερά *bsIncr* και τις (στατικές) βοηθητικές συναρτήσεις *cStrLen, stringCmpr*.

Ας δούμε πώς θα γραφεί η επιφόρτωση του "`+=`":

```
BString& BString::operator+=( const BString& rhs )
{
   if ( bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 >
                                          bsHandle->bsReserved )
   {
      char* tmp;
      size_type tmpRes( ((bsHandle->bsLen+(rhs.bsHandle)->bsLen+1)/
                     BStringImpl::bsIncr+1)*BStringImpl::bsIncr );
      try {  tmp = new char[tmpRes];  }
      catch( bad_alloc& )
      {  throw BStringXptn( "operator+=",
                           BStringXptn::allocFailed );  }
      for ( int k(0); k < bsHandle->bsLen; ++k )
         tmp[k] = bsHandle->bsData[k];
      delete[] bsHandle->bsData;
      bsHandle->bsData = tmp;
   }
   for ( int j(0), k(bsHandle->bsLen);
         j < (rhs.bsHandle)->bsLen;
         ++j, ++k )
      bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];
   bsHandle->bsLen += (rhs.bsHandle)->bsLen;
   return *this;
} // BString::operator+=
```

Όπως βλέπεις:

- Στη συνθήκη της **if** αντί για

  `bsLen + rhs.bsLen + 1 > bsReserved`

  γράφουμε

  `bsHandle->bsLen + (rhs.bsHandle)->bsLen + 1 > bsHandle->bsReserved`
- Στην αρχική τιμή της *tmpRes* γράφουμε "`BStringImpl::bsIncr`" και όχι `bsIncr`.
- Στη δεύτερη **for** αντιγράφουμε με τη

  `bsHandle->bsData[k] = (rhs.bsHandle)->bsData[j];`

και όχι με τη

```
bsData[k] = rhs.bsData[j];
```

Τα ίδια ισχύουν και για τις επιφορτώσεις των τελεστών "`==`" και "`<<`" που τώρα τις έχουμε δηλώσει "`friend`":[13]

```
ostream& operator<<( ostream& tout, const BString& rhs )
{
   for ( int k(0); k < (rhs.bsHandle)->bsLen; ++k )
      tout << (rhs.bsHandle)->bsData[k];
   return tout;
} // operator<<( ofstream, BString

bool operator==( const BString& lhs, const BString& rhs )
{
   int fv( BString::BStringImpl::stringCmpr(
                                        (lhs.bsHandle)->bsData,
                                        (rhs.bsHandle)->bsData,
            min((lhs.bsHandle)->bsLen, (rhs.bsHandle)->bsLen)) );
   if ( fv == 0 )
      fv = (lhs.bsHandle)->bsLen - (rhs.bsHandle)->bsLen;
   return ( fv == 0 );
} // operator==
```

Ακολουθώντας αυτά που είπαμε στην §18.3 μπορούμε από το `BString.cpp` (και το `BString.h`) να πάρουμε το μεταγλωττισμένο αρχείο `BString.o` (ή `BString.obj`).

Τώρα το `BString.h` είναι:

```
#ifndef _BSTRING_H
#define _BSTRING_H

#include <fstream>
#include <string>

using namespace std;

class BString
{
friend bool operator==( const BString& lhs, const BString& rhs );
friend ostream& operator<<( ostream& tout, const BString& rhs );
public:
   BString( const char* rhs="" );
   BString( const BString& rhs );
   ~BString();
   BString& operator=( const BString& rhs );
   BString& assign( const BString& rhs );
   const char* c_str() const;
   size_type length() const;
   bool empty() const;
   char& at( int k ) const;
   char& operator[]( int pos ) const;
   BString& operator+=( const BString& rhs );
   BString& append( const BString& rhs );
   void swap( BString& rhs );
   int compare( const BString& rhs ) const;
private:
   struct BStringImpl;
   BStringImpl* bsHandle;
}; // BString

BString operator+( const BString& lhs, const BString& rhs );

struct BStringXptn
{
   enum { allocFailed, outOfRange };
```

[13] Η επιφόρτωση του "+" παραμένει όπως δόθηκε στην §22.6.1.

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

#endif // _BSTRING_H
```

Ένας προγραμματιστής, έχοντας το **BString.o** (ή **.obj**) και το **BString.h**, *μπορεί να χρησιμοποιήσει στα προγράμματά του τη BString χωρίς να ξέρει πώς είναι γραμμένη.*

Και ακόμη: αν εμείς αλλάξουμε την υλοποίηση της κλάσης χωρίς να αλλάξουμε τη διεπαφή (δηλαδή αυτά που έχουμε στο "**public**" και τις επικεφαλίδες των καθολικών συναρτήσεων) οποιοδήποτε πρόγραμμα χρησιμοποιεί την κλάση δεν χρειάζεται να περάσει ξανά από τον μεταγλωττιστή· αρκεί να περάσει από τον συνδέτη.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

22-1 Επιφόρτωσε τους τελεστές "**-**", "**--**" και "**-=**" για τη *Date* ώστε να δίνουν αντιστοίχως τη διαφορά (σε ημέρες) δύο ημερομηνιών και την «οπισθοδρόμηση» μιας ημερομηνίας.

22-2 Επιφόρτωσε όλους τους τελεστές σύγκρισης για τις *Date* και *BString*.