κεφάλαιο

23

Κληρονομιές

Ο στόχος μας σε αυτό το κεφάλαιο:
Θα γνωρίσουμε την έννοια της παραγωγής μιας κλάσης από μια άλλη και της κληρονομιάς που παίρνει η νέα από την παλιά.
Προσδοκώμενα αποτελέσματα:
Εξοικείωση με την κλήρονομικότητα και τις σχετικές προγραμματιστικές τεχνικές.
Έννοιες κλειδιά:

- *βασική κλάση*
- *παράγωγη κλάση*
- *σχέση* `is_a`
- *προστατευόμενα (*`protected`*) μέλη*
- *εικονικές (*`virtual`*) μέθοδοι*
- *πολυμορφισμός*
- *δυναμική τυποθεώρηση*
- *εξακρίβωση τύπου κατά την εκτέλεση -- RTTI*

Περιεχόμενα:

23.1 Κτίζοντας Πάνω στα Υπάρχοντα ... 842
23.1.1 Τι ΔΕΝ Κληρονομείται ... 843
23.2 Σχέσεις Αντικειμένων και Κλάσεων ... 844
23.3 Ο Νέος Δημιουργός ... 846
23.3.1 Και Ένας Άλλος Τρόπος ... 848
23.3.2 Ο Δημιουργός Αντιγραφής ... 848
23.4 Και ο Καταστροφέας ... 849
23.5 Νέες και Παλιές Μέθοδοι ... 849
23.5.1 Ο Τελεστής Εκχώρησης ... 852
23.6 Καθολικές Συναρτήσεις ... 853
23.7 Κλάση Εξαιρέσεων Παράγωγης Κλάσης ... 853
23.8 "`private`" ή "`protected`" ... 854
23.8.1 * "`protected`": Ψιλά Γράμματα ... 855
23.9 Εικονικές Μέθοδοι ... 857
23.10 * Υλοποίηση Εικονικών Συναρτήσεων ... 859
23.10.1 "`virtual`" και Τεμαχισμός ... 860
23.10.2 Κάποια Σχόλια... ... 861
23.11 Εικονικός Καταστροφέας ... 861
23.12 Περί Πολυμορφισμού ... 862
23.12.1 Πολυμορφισμός Χρόνου Μεταγλώττισης(;) ... 863
23.12.2 * Υπερίσχυση ή Επιφόρτωση ... 864
23.13 Δυναμική Τυποθεώρηση - RTTI ... 866
23.13.1 Μετατροπή Αναφορών ... 868
23.14 "`is_a`" ή "`has_a`" ... 869

23.15 "private", "protected" και "public" 872
23.16 Αφηρημένες Κλάσεις 874
23.17 Η Σειρά Δημιουργίας 875
23.18 Πολλαπλή Κληρονομιά 876
23.19 Διεπαφές 878
23.20 Ανακεφαλαίωση 879
Ερωτήσεις - Ασκήσεις 879
Α Ομάδα 879
Β Ομάδα 880

Εισαγωγικές Παρατηρήσεις:
Ένα βασικό χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού είναι η κληρονομικότητα: η δυνατότητα μιας κλάσης να δηλώνεται με βάση μιαν άλλη κλάση που προϋπάρχει και να έχει ως **κληρονομιά** (inheritance) τα μέλη και τις μεθόδους της αρχικής.

Στη συνέχεια θα αποφύγουμε τις πολλές θεωρίες και δούμε τα περί κληρονομιάς με παραδείγματα σε C++ (αφού, καλώς ή κακώς, οι τρόποι υλοποίησης των βασικών εννοιών στη C++ έχουν το ίδιο μεγάλο ενδιαφέρον με τις ίδιες τις έννοιες.)

23.1 Κτίζοντας Πάνω στα Υπάρχοντα

Η ημερομηνία ως χρονικός προσδιορισμός δεν είναι πάντοτε επαρκής. Οι περιπτώσεις που χρειαζόμαστε μεγαλύτερη ακρίβεια είναι πολλές. Αφού η *Date* δεν επαρκεί ας γράψουμε μια νέα κλάση, ας την πούμε *DateTime*, που θα πηγαίνει μέχρι δευτερόλεπτο:

```
class DateTime
{
public:
// . . .
private:
   unsigned int  dtYear;
   unsigned char dtMonth;
   unsigned char dtDay;
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Τι θα κάνουμε με τα *dtYear*, *dtMonth*, *dtDay*; Αυτό μπορούμε να το αντιγράψουμε από τη *Date*, αλλά δεν χρειάζεται! Δες έναν άλλον τρόπο να γράψουμε τα παραπάνω με πολύ «περισσότερο περιεχόμενο»:

```
class DateTime: public Date
{
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
// . . .
private:
   unsigned char dtHour;    // hour (0 .. 23)
   unsigned char dtMin;     // minutes (0 .. 59)
   unsigned char dtSec;     // seconds (0 .. 59)
}; // DateTime
```

Αφήνοντας κατά μέρος τον δημιουργό, πού βρίσκεται το «περισσότερο περιεχόμενο»; Μας το δείχνει το παρακάτω προγραμματάκι:

```
   DateTime dt( 2007, 11, 26, 19, 30, 31 );
   cout << dt.getDay() << '.' << dt.getMonth() << '.' << dt.getYear() << endl;
   cout << dt << endl;
   dt += 3;          cout << dt << endl;
   dt.forward( 3 );  cout << dt << endl;
   ++dt;             cout << dt << endl;
   cout << (dt < Date(2008)) << endl;
```

που μας δίνουν:

```
26.11.2007
26.11.2007
26.11.2007
26.11.2007
27.11.2007
1
```

Η *dt* δηλώθηκε μεν κλάσης *DateTime* αλλά συμπεριφέρεται πλήρως ως αντικείμενο κλάσης *Date*! Ή αλλιώς:

- Η κλάση *DateTime* **κληρονομεί** (inherits) –για κάθε αντικείμενό της– όλα τα χαρακτηριστικά και τη συμπεριφορά που έχει ένα αντικείμενο της *Date*.
- Αλλά, κάθε αντικείμενό της *DateTime* έχει επιπλέον χαρακτηριστικά και μπορούμε να «εμπλουτίσουμε» τη συμπεριφορά του με επιπλέον μεθόδους. Αυτό και θα κάνουμε στη συνέχεια.

Αφού η *DateTime* κληρονομεί τη συμπεριφορά της *Date* θα κληρονομεί και την αναλλοίωτη. Πράγματι, ας δούμε την αναλλοίωτη της νέας κλάσης:

$$I_{DateTime}: (dYear > 0)\ \&\&\ (0 < dMonth <= 12)\ \&\&\ (0 < dDay <= lastDay(dYear, dMonth))$$
$$\&\&\ (0 < dtHour < 24)\ \&\&\ (0 < dtMin < 60)\ \&\&\ (0 < dtSec < 60)$$

Ή

$$I_{DateTime}: I_{Date}\ \&\&\ (0 < dtHour < 24)\ \&\&\ (0 < dtMin < 60)\ \&\&\ (0 < dtSec < 60)$$

Δηλαδή, αφού η I_{Date} συνδέεται με τα υπόλοιπα με "**&&**":

- ***Ό,τι ισχύει για τα αντικείμενα της βασικής κλάσης ισχύει και για τα αντικείμενα της παράγωγης κλάσης.***

Αυτή είναι η **αρχή της Liskov** (Liskov & Wing, 1993).

«*Βασική*»; «*Παράγωγη*»; Ήλθε η ώρα για λίγη ορολογία: Λέμε ότι η κλάση *DateTime* **παράγεται** (is derived) από την *Date* και παίρνει ως **κληρονομιά** (inheritance) όλα τα μέλη και τις μεθόδους της. Λέμε ότι η (κληρονομούμενη) κλάση *Date* είναι η

- **βασική κλάση** (base class) ή
- **υπερκλάση** (superclass) ή
- **γονική** (parent) κλάση

ενώ η κληρονόμος *DateTime* ονομάζεται

- **παράγωγη** (derived) κλάση ή
- **υποκλάση** (subclass) ή
- **κλάση-παιδί** (child class).

23.1.1 Τι ΔΕΝ Κληρονομείται

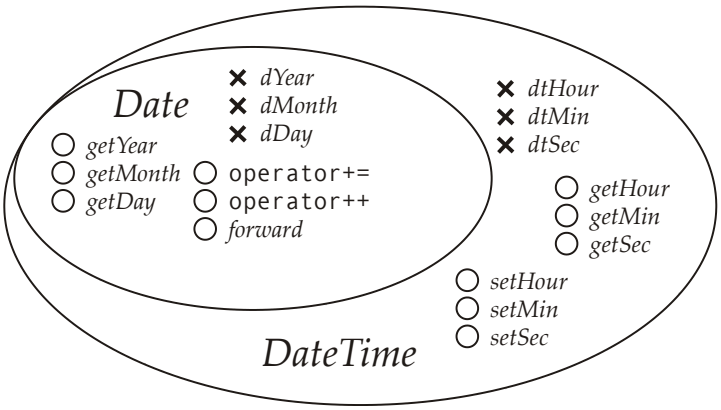
Γενικώς:

- ***Οι δημιουργοί δεν κληρονομούνται.***

με την έννοια που είδαμε στο παράδειγμα. Πέρα από αυτό όμως, υπάρχουν τέσσερις «ειδικές συναρτήσεις»:

- ο ερήμην δημιουργός,
- ο δημιουργός αντιγραφής,
- ο τελεστής εκχώρησης,
- ο καταστροφέας,

που όχι μόνο δεν κληρονομούνται με την έννοια που είδαμε πιο πάνω, αλλά αν δεν ορίσουμε κάποια από αυτές, όταν ορίζουμε μια κλάση, ο μεταγλωττιστής θα ορίσει αυτομάτως μια *συναγόμενη* (implicit). Αυτό, όπως ξέρουμε, δεν σημαίνει ότι η συναγόμενη συνάρτηση



Σχ. 23-1 Κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

θα συμπεριφέρεται όπως τη θέλουμε. Για τον λόγο αυτόν ακριβώς έχουμε βάλει κανόνες για το πότε πρέπει να παρεμβαίνουμε.

Τα παραπάνω ισχύουν και για τις παράγωγες κλάσεις: *Αν δεν ορίσουμε οποιαδήποτε από τις παραπάνω συναρτήσεις ο μεταγλωττιστής θα ορίσει αυτομάτως μια συναγόμενη.*

Αν όμως θελήσουμε να ορίσουμε δικές μας συναρτήσεις, μπορούμε να χρησιμοποιήσουμε τις αντίστοιχες της βασικής κλάσης όπως θα δούμε στη συνέχεια.

Τέλος, να πούμε ότι:

- ***Οι φιλίες δεν κληρονομούνται.***

Δηλαδή, το ότι μια συνάρτηση ή μια κλάση είναι φίλη της βασικής κλάσης δεν σημαίνει ότι αυτομάτως είναι και φίλη της παράγωγης. Αν θέλεις να είναι φίλη της παράγωγης θα πρέπει να το δηλώσεις.

23.2 Σχέσεις Αντικειμένων και Κλάσεων

Δες το Σχ. 23-1: να πώς περίπου είναι ένα αντικείμενο κλάσης *DateTime*. Όπως βλέπεις περιέχει ένα «υποαντικείμενο» κλάσης *Date*.

Πώς ερμηνεύονται λοιπόν αυτά που είδαμε στο προηγούμενο παράδειγμα; Δηλαδή: γιατί ένα αντικείμενο κλάσης *DateTime* συμπεριφέρεται σαν αντικείμενο κλάσης *Date*; Διότι κάθε αντικείμενο κλάσης *DateTime* περιέχει ένα «υποαντικείμενο» κλάσης *Date* που ανταποκρίθηκε με τον τρόπο που είδαμε σε όλα τα μηνύματα που του στείλαμε.

Γενικεύοντας δίνουμε δύο κανόνες:

- ***Κάθε αντικείμενο της παράγωγης κλάσης D περιέχει ένα υποαντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

  Λόγω αυτού του γεγονότος:

- ***Κάθε αντικείμενο της παράγωγης κλάσης D συμπεριφέρεται σαν αντικείμενο της βασικής κλάσης B την οποίαν η D κληρονομεί.***

Και ακόμα γενικότερα:

- ***Οπουδήποτε, μέσα στο πρόγραμμά μας μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D.***

Αυτό λέγεται **δυνατότητα υποκατάστασης** (substitutability) και είναι μια άλλη θεώρηση της αρχής της Liskov (Liskov Substitution Principle - LSP).[1]

[1] Στο τέλος του κεφαλαίου θα δεις μια ακριβέστερη διατύπωση.

Για παράδειγμα, αν έχουμε δηλώσει:

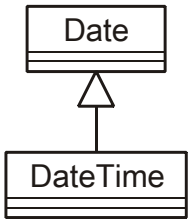
```
Date d;
DateTime dt;
Date* pD;
DateTime* pDt;
```

είναι νόμιμες οι εντολές:

```
d = dt;       // στη d εκχωρείται
              // το υποαντικείμενο κλάσης Date της dt
pD = new DateTime;
```

αλλά δεν είναι νόμιμες οι:[2]

```
dt = d;
pDt = new Date;
```



Σχ. 23-2 Παράσταση της συσχέτισης *DateTime* `is_a` *Date*.

Το ότι με τη "`d = dt`" εκχωρείται στη *d* το υποαντικείμενο κλάσης *Date* της *dt* ονομάζεται **τεμαχισμός** (slicing) του αντικειμένου. Αυτό θα το ξαναδούμε...

Πρόσεξε όμως και το εξής: Αν δούμε τις κλάσεις ως σύνολα αντικειμένων τότε *η βασική κλάση B περιέχει και τα αντικείμενα της παράγωγης κλάσης D*, δηλαδή η *D* είναι υποσύνολο της *B* (ενώ το αντικείμενο της *B* είναι «υποσύνολο» του αντικειμένου της *D*). Αντί για $D \subseteq B$ γράφουμε *D* `is_a` *B* και δίνουμε μια διαγραμματική παράσταση σαν αυτήν του Σχ. 23-2. Το βέλος δείχνει από την παράγωγη προς τη βασική. Το νόημά του είναι «*η D **αναφέρεται προς** (refers to) την B*».

Σημείωση:▶
Ορισμένοι αντί για *D* `is_a` *B* γράφουν *D* `a_kind_of` *B* και κρατούν την `is_a` ως *σχέση μεταξύ των αντικειμένων* των δύο κλάσεων. Εμείς χρησιμοποιούμε το `is_a` και για τις δύο περιπτώσεις.◀

Στη συνέχεια θα δούμε πώς μπορούμε να χειριζόμαστε το υποαντικείμενο της βασικής κλάσης μέσα στις μεθόδους της παράγωγης. Προς το παρόν να πούμε ότι μπορείς να φτάσεις σε αυτό ως εξής:

- Το `this` είναι βέλος προς αντικείμενο την παράγωγης κλάσης, στο παράδειγμά μας τύπου *DateTime*`*`.
- Η παράσταση "`static_cast<Date*>(this)`" μας δίνει ένα αντίγραφο του `this` αλλά τύπου *Date*`*`. Αυτό *είναι βέλος προς το υποαντικείμενο κλάσης Date*`*`.
- Το "`*(static_cast<Date*>(this))`" είναι *το υποαντικείμενο κλάσης Date*.

Να και ένα σχετικό

Παράδειγμα
Η "`dt = d`" απαγορεύεται με την έννοια ότι δεν υπάρχει παρόμοια *συναγόμενη* επιφόρτωση του "`=`". Φυσικά, αν την ορίσουμε δεν υπάρχει πρόβλημα.

Αν ορίσουμε, για παράδειγμα τον *δημιουργό μετατροπής*:

```
DateTime& DateTime::operator=( const Date& rhs )
{
   *(static_cast<Date*>(this)) = rhs:
   dtHour = 12;  dtMin = 0;  dtSec = 0;
} // DateTime::operator=
```

–που είναι κάτι λογικό– μια τέτοια εκχώρηση γίνεται νόμιμη.
&

Έξω από το αντικείμενο της παράγωγης κλάσης. μπορείς να πάρεις ένα αντίγραφο του υποαντικειμένου της βασικής με στατική τυποθεώρηση. Για παράδειγμα, το

"`static_cast<Date>(dt)`"

θα σου δώσει ένα αντίγραφο του υποαντικειμένου κλάσης *Date* του *dt*.

Η "`d = dt`", που είδαμε, εκτελείται ως "`d = static_cast<Date>(dt)`".

---

[2] Αργότερα θα δούμε πώς μπορούμε να κάνουμε νόμιμες και αυτές τις εντολές.

23.3 Ο Νέος Δημιουργός

Ας πούμε ότι, λόγω εκτέλεσης μιας δήλωσης, δημιουργείται αντικείμενο που αποτελείται από άλλα αντικείμενα είτε λόγω κληρονομιάς είτε λόγω δήλωσης μελών. Η σειρά δημιουργίας αντικειμένου που είδαμε στην §21.3 χρειάζεται συμπλήρωση[3]:

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται.
- Εκτελείται το σώμα του δημιουργού.

Παράδειγμα

Στο πρόγραμμα:

```
 0:   #include <iostream>
 1:   using namespace std;
 2:
 3:   class A
 4:   {
 5:      int ma;
 6:   public:
 7:      A( int ap = 0 )
 8:      {  ma = ap;  cout << "A object created" << endl;  }
 9:   }; // A
10:
11:   class B
12:   {
13:      int mb;
14:   public:
15:      B( int bp = 0 )
16:      {  mb = bp;  cout << "B object created" << endl;  }
17:   }; // B
18:
19:   class C
20:   {
21:      int mc;
22:   public:
23:      C( int cp = 0 )
24:      {  mc = cp;  cout << "C object created" << endl;  }
25:   }; // C
26:
27:   class D: public B
28:   {
29:      A da;
30:      C dc;
31:   public:
32:      D() {  cout << "D object created" << endl;  }
33:   }; // D
34:
35:   int main()
36:   {
37:      D d;
38:   }
```

βλέπουμε ότι η κλάση *D* (γρ. 27-33) είναι παράγωγη της *B* (γρ. 11-17) και έχει δύο μέλη, το πρώτο (γρ. 29) κλάσης *A* (γρ. 3-9), το δεύτερο (γρ. 30) κλάσης *C* (γρ. 19-25). Με βάση αυτά που είπαμε, για να δημιουργηθεί το αντικείμενο *d* (γρ. 37 στη `main`)

- Πρώτα θα κληθεί ο δημιουργός της *B*, για να δημιουργήσει το υποαντικείμενο κλάσης *B* που θα υπάρχει στο *d*.
- Μετά θα δημιουργηθούν τα μέλη *da* και *dc* από τους δημιουργούς των κλάσεων *A* και *C*.
- Τέλος, θα εκτελεσθεί το σώμα του δημιουργού της *D*.

  Αυτά επιβεβαιώνονται από τα αποτελέσματα:

[3] Ούτε τώρα είναι πλήρης ο κανόνας. Θα επανέλθουμε...

```
B object created
A object created
C object created
D object created
```

Να γράψουμε λοιπόν για την *DateTime* τον δημιουργό:

```
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 );
```

Σύμφωνα με τα παραπάνω, όταν θα εκτελείται, αρχικώς θα δημιουργηθεί το υποαντικείμενο της βασικής κλάσης, της *Date*, μετά θα δημιουργηθούν τα μέλη *dtHour*, *dtMin*, *dtSec* και τέλος θα εκτελεσθεί το σώμα όπου θα δώσουμε τιμές στα μέλη (αφού χρειάζονται και έλεγχοι):

```
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Καλά όλα αυτά, αλλά στα μέλη του υποαντικειμένου πώς θα δώσουμε τιμές; Αν προσπαθήσουμε να γράψουμε

```
   dYear = yp;  dMonth = mp;  dDay = dp;
```

μας περιμένει μια δυσάρεστη έκπληξη[4]:

```
Error E2247 dateTime01.cpp 128: 'Date::dYear' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dMonth' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
Error E2247 dateTime01.cpp 128: 'Date::dDay' is not accessible in function
DateTime::DateTime(int,int,int,int,int,int)
```

Δηλαδή: ο δημιουργός της *DateTime* δεν έχει πρόσβαση στα *dYear*, *dMonth*, *dDay* του υποαντικειμένου *Date*! Για όλα φταίει εκείνο το "`private:`" που έχουμε στη *Date* και στη συνέχεια θα δούμε πώς διορθώνεται.

Εδώ θα δώσουμε άλλη λύση στο πρόβλημά μας, με τη λίστα εκκίνησης:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Τι λέμε εδώ; Δημιούργησε το υποαντικείμενο κλάσης *Date* με παραμέτρους *yp*, *mp*, *dp* και μετά συμπλήρωσε τις τιμές των «νέων» μελών. Με αυτόν τον τρόπο:

[4] Borland C++ v.5.5

- Δεν χρειάζεται να ξαναβάλουμε τους ελέγχους αφού θα τους κάνει ο δημιουργός της *Date*.
- Τα μέλη του υποαντικειμένου *Date* παίρνουν τιμή με τη δημιουργία του. Δεν χρειάζεται να πάρουν τις ερήμην τιμές και μετά να βάλουμε αυτές που πιθανόν έχουν δοθεί.

Βλέπουμε λοιπόν ότι *στη λίστα εκκίνησης μιας παράγωγης κλάσης μπορούμε να καλούμε τον δημιουργό της βασικής για να δημιουργήσει το αντίστοιχο υποαντικείμενο.*

23.3.1 Και Ένας Άλλος Τρόπος

Με βάση αυτά που είπαμε για εκχώρηση τιμής στο υποαντικείμενο της βασικής κλάσης μπορούμε να γράψουμε:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::hourRange, hp );
   if ( minp < 0 || 59 < minp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::minRange, minp );
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "DateTime",
                          DateTimeXptn::secRange, sp );
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
   dtHour = hp;  dtMin = minp;  dtSec = sp;
}; // DateTime::DateTime
```

Φυσικά, δεν εννοούμε να προτιμήσεις αυτόν τον τρόπο. Ο πρώτος είναι σαφώς καλύτερος.

23.3.2 Ο Δημιουργός Αντιγραφής

Κατ' αρχάς να ξεκαθαρίσουμε ότι ο *εννοούμενος δημιουργός αντιγραφής της DateTime είναι σωστός*.

Θα γράψουμε όμως εδώ έναν τέτοιο δημιουργό μόνο και μόνο για παράδειγμα. Αλλά, πριν από αυτό θα δώσουμε ένα παράδειγμα της *δυνατότητας υποκατάστασης*. Αν έχουμε ορίσει:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
```

τότε οι

```
   Date d( dt );
   cout << d << endl;
```

θα δώσουν:

```
16.11.2007
```

Δηλαδή: Ένα αντικείμενο της παράγωγης κλάσης μπορεί να χρησιμοποιηθεί ως αρχική τιμή για ένα αντικείμενο της βασικής. Φυσικά, έχουμε τεμαχισμό και στο *d* εκχωρείται η τιμή του υποαντικειμένου κλάσης *Date* του *dt*.

Με βάση αυτό, να πώς μπορούμε να γράψουμε τον δημιουργό αντιγραφής της *DateTime*:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs )
{
   dtHour = rhs.dtHour;  dtMin = rhs.dtMin;  dtSec = rhs.dtSec;
}; // DateTime::DateTime
```

Με το "`Date(rhs)`" καλούμε τον δημιουργό αντιγραφής της *Date* για να δημιουργήσει αντίγραφο του υποαντικειμένου *Date* του *rhs*. Το σώμα του δημιουργού αναλαμβάνει την αντιγραφή των άλλων μελών.[5]

Από τον δημιουργό αντιγραφής παίρνουμε τον τελεστή εκχώρησης όπως μάθαμε στην §21.7, αφού προηγουμένως ορίσουμε την «ασφαλή *swap*». Το βλέπουμε στη συνέχεια.

23.4 Και ο Καταστροφέας ...

Η καταστροφή του αντικειμένου της παράγωγης κλάσης γίνεται με την αντίστροφη σειρά από αυτήν της δημιουργίας:

- Εκτελείται το σώμα του καταστροφέα.
- Καταστρέφονται τα μέλη του αντικειμένου.
- Καλείται (αυτομάτως) ο καταστροφέας της βασικής κλάσης και καταστρέφει το υποαντικείμενο της βασικής κλάσης.

Για παράδειγμα, αν στο πρόγραμμα της §23.3 εφοδιάσεις τις κλάσεις σου με κατάστροφείς του είδους:

```
   ~A() {  cout << "destroying A object" << endl;  }
   ~B() {  cout << "destroying B object" << endl;  }
 . . .
```

θα δεις στο τέλος της εκτέλεσης:

```
destroying D object    (εκτελείται το σώμα του καταστροφέα)
destroying C object    (καταστρέφονται τα μέλη του αντικειμένου)
destroying A object
destroying B object    (καταστρέφεται το υποαντικείμενο της βασικής)
```

Με τον καταστροφέα όμως δεν τελειώσαμε· θα επανέλθουμε...

23.5 Νέες και Παλιές Μέθοδοι

Να δούμε τι άλλο χρειαζόμαστε για τη νέα κλάση εκτός από τον δημιουργό και αυτά που κληρονομούνται από βασική.

Σίγουρα θα χρειαστούμε μεθόδους "*get*" και "*set*" για τα τρία μέλη:

```
   unsigned int getHour() const {  return dtHour;  };
   unsigned int getMin() const {  return dtMin;  };
   unsigned int getSec() const {  return dtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
```

όπου:

```
void DateTime::setHour( int hp )
{
   if ( hp < 0 || 23 < hp )
      throw DateTimeXptn( "setHour",
                          DateTimeXptn::hourRange, hp );
   dtHour = hp;
} // DateTime::setHour

void DateTime::setMin( int minp )
{
   if ( minp < 0 || 59 < minp )
```

[5] Αυτά γενικώς. Στην συγκεκριμένη περίπτωση θα ήταν καλύτερο να γράψουμε:

```
DateTime::DateTime( const DateTime& rhs )
   :  Date( rhs ), dtHour( rhs.dtHour ), dtMin( rhs.dtMin ), dtSec( rhs.dtSec ) { }
```

```
      throw DateTimeXptn( "setMin",
                          DateTimeXptn::minRange, minp );
   dtMin = minp;
} // DateTime::setMin

void DateTime::setSec( int sp )
{
   if ( sp < 0 || 59 < sp )
      throw DateTimeXptn( "setSec",
                          DateTimeXptn::minRange, sp );
   dtSec = sp;
} // DateTime::setSec
```

Μια μέθοδος που μας ενδιαφέρει είναι η *forward()* –ή ο τελεστής "+="– αλλά τη θέλουμε *να αυξάνει τα δευτερόλεπτα και όχι τις ημέρες*. Γίνεται; Γίνεται:[6]

```
DateTime& DateTime::forward( long int ds )
{
   tm     currD = { dtSec, dtMin, dtHour, getDay(),
                    getMonth()-1, getYear()-1900, 0, 0, 0 };
   time_t currT( mktime(&currD) );
   if ( ds < 0 )
   {
      if ( currT < (-ds) )  // currT + ds < 0
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   else // ds >= 0
   {
      if ( ds > LONG_MAX - currT )  // currT + ds > ULONG_MAX
         throw DateTimeXptn( "operator+=",
                             DateTimeXptn::outOfLimits,
                             ds, *this );
   }
   currT += ds;
   currD = *localtime( &currT );
   *(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                       currD.tm_mon+1,
                                       currD.tm_mday );
   dtHour = currD.tm_hour;
   dtMin = currD.tm_min;
   dtSec = currD.tm_sec;
   return *this;
} // Date::forward
```

Πρόσεξε δύο σημεία:

- Χρησιμοποιούμε και εδώ τον "=" της βασικής κλάσης (*Date*).
- Πρόσεξε τις επικεφαλίδες της μεθόδου στις δύο κλάσεις:

```
Date& forward( long int dd )
DateTime& forward( long int ds )
```

Διαφέρουν μόνο στον τύπο της επιστρεφόμενης τιμής. Δες όμως το αποτέλεσμα αυτής της ομοιότητας:

```
   DateTime  dt( 2007, 11, 16, 19, 30, 31 );
   Date d( dt );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
   d.forward( 10 );
   dt.forward( 10 );
   cout << d << endl;
   cout << dt.getDay() << '.' << dt.getMonth() << '.'
```

[6] Αν δεν καταλαβαίνεις τις λεπτομέρειες της υλοποίησης ξαναδιάβασε την §22.5.1.4.

```
        << dt.getYear() << ' ' << dt.getHour() << ':'
        << dt.getMin() << ':' << dt.getSec() << endl;
```

Αποτέλεσμα:

```
16.11.2007
16.11.2007 19:30:31
26.11.2007
16.11.2007 19:30:41
```

Αν ορίσουμε:

```
DateTime& operator+=( long int ds ) {  return forward( ds );  }
```

και αλλάξουμε τίς κλήσεις προς τις *forward()* σε:

```
d += 10;
dt += 10;
```

θα πάρουμε τα ίδια αποτελέσματα.

Και στις δύο περιπτώσεις, δράσαμε με την ίδια μέθοδο (τον ίδιο τελεστή) και το ίδιο όρισμα (10) και στα δύο αντικείμενα –*d* κλάσης *Date* και *dt* κλάσης *DateTime*– και η πρώτη προχώρησε κατά 10 ημέρες ενώ η δεύτερη προχώρησε κατά 10 *sec*. Τέτοιες μέθοδοι (τελεστές) λέγονται *πολυμορφικές*. Ο πολυμορφισμός από κληρονομιές βασίζεται στο ότι, για τα αντικείμενα της παράγωγης κλάσης η μέθοδος της παράγωγης κλάσης **υπερισχύει** (overrides) της μεθόδου της βασικής.

- ***Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.***

Δηλαδή, δεν υπάρχει περίπτωση να χρησιμοποιήσει ένα αντικείμενο της *DateTime* την *forward()* της *Date*; Ναι, αν ζητηθεί πιο συγκεκριμένα. Στο παραπάνω παράδειγμα δίνουμε:

```
dt.Date::operator+=( 10 );
```

ή

```
dt.Date::forward( 10 );
```

και η τιμή του *dt* γίνεται:

```
26.11.2007 19:30:41
```

Με τον πολυμορφισμό δεν τελειώσαμε. Στη συνέχεια θα δούμε μερικές πολύ ενδιαφέρουσες πτυχές του θέματος.

Παρατήρηση:▶

Με το "`this->Date::`" μπορούμε να καλούμε τις μεθόδους του υποαντικειμένου κλάσης *Date* και μέσα στις μεθόδους της παράγωγης κλάσης (*DateTime*). Για παράδειγμα αντί για

```
*(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                    currD.tm_mon+1,
                                    currD.tm_mday );
```

θα μπορούσαμε να γράψουμε:

```
this->Date::operator=( Date(currD.tm_year+1900, currD.tm_mon+1,
                            currD.tm_mday) );
```

Με αυτόν τον τρόπο δεν δημιουργείται αντίγραφο του `this` και έχεις ένα (πολύ μικρό) κέρδος.◀

Ας δούμε τώρα ένα άλλο πρόβλημα: Αν κάποιος γράψει `++dt` τότε η τιμή της *dt* θα προχωρήσει κατά μια ημέρα. Αυτό μπορεί να δημιουργήσει σύγχυση. Να την ξαναορίσουμε ώστε να προχωράει κατά 1 *sec*; Ας πούμε ότι αυτό δεν μας ενδιαφέρει· αν θέλουμε να προχωρήσουμε 1 *sec* θα χρησιμοποιούμε τη *forward()*. Τι κάνουμε;

Ορίζουμε στην περιοχή `private` της *DateTime*:

```
DateTime& operator++() { }
```

Αν τυχόν σου ξεφύγει κάποιο `++dt` μέσα στο πρόγραμμά σου θα το φροντίσει ο μεταγλωττιστής: «`'DateTime& operator++()' is private`.»

Φυσικά, αν κάποτε θελήσεις να χρησιμοποιήσεις τον "`++`" της *Date* μπορείς να δώσεις:

```
dt.Date::operator++();
```

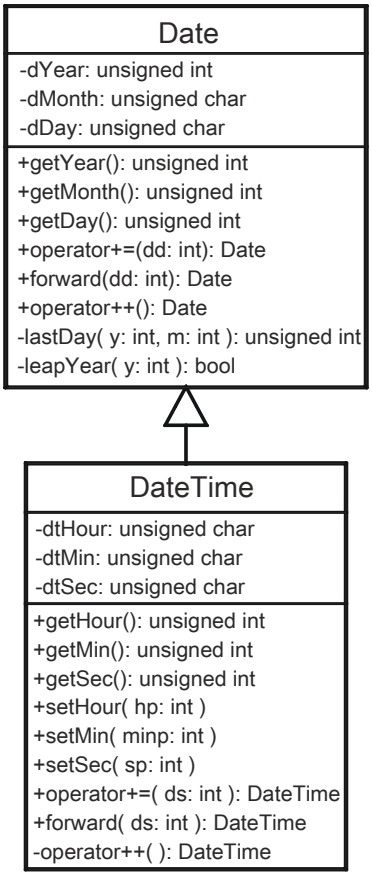
Παρατήρηση: ▶

Αν εσύ επιμένεις να επιφορτώσεις τον "`++`" ώστε η "`++dt`" να αυξάνει τον χρόνο κατά 1 *sec* θα πρέπει, σύμφωνα με αυτά που είπαμε στην §22.2, να την ορίσεις **inline** με βάση τη *forward*:

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

Αυτός ο ορισμός διαφέρει από τον αντίστοιχο της *Date* (§22.5.1.6) μόνον στον τύπο του αποτελέσματος.◀

Στο Σχ. 23-3 βλέπεις ένα λεπτομερέστερο διάγραμμα κληρονομιάς.



Σχ. 23-3 Η κλάση *DateTime* είναι παράγωγη της κλάσης *Date*. Κληρονομεί από αυτήν τα *dYear*, *dMonth*, *dDay*, *getYear*, *getMonth*, *getDay*. Δεν κληρονομεί τις `operator+=`, *forward*, `operator++` (έχει δικές της).

23.5.1 Ο Τελεστής Εκχώρησης

Αν χρειάζεται να ορίσεις τον τελεστή εκχώρησης για την παράγωγη κλάση όρισέ τον με βάση τον δημιουργό αντιγραφής που, σύμφωνα με τον «κανόνα των τριών», μάλλον θα πρέπει να έχεις ορίσει. Χρησιμοποιώντας το πάγιο «πατρόν», δεν έχεις παρά να ορίσεις τη *swap()*.

Ορίζουμε τη *swap()* χρησιμοποιώντας τη *swap()* της βασικής. Στο παράδειγμά μας:

```
void DateTime::swap( DateTime& rhs )
{
   this->Date::swap( rhs );
   std::swap( dtHour, rhs.dtHour );
   std::swap( dtMin, rhs.dtMin );
   std::swap( dtSec, rhs.dtSec );
} // DateTime::swap
```

Η "`this->Date::swap(rhs)`" (ή, επί το απλούστερο: "`Date::swap(rhs)`") ασχολείται μόνο με το υποαντικείμενο *Date* του *rhs*.[7]

Μετά ο δρόμος είναι γνωστός:

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

[7] Μα δεν έχουμε ορίσει *swap()* για τη *Date*! Ας την ορίσουμε:

```
void Date::swap( Date& rhs )
{
   std::swap( dYear, rhs.dYear );
   std::swap( dMonth, rhs.dMonth );
   std::swap( dDay, rhs.dDay );
} // Date::swap
```

23.6 Καθολικές Συναρτήσεις

Όπως στις μεθόδους, έτσι και στην υλοποίηση καθολικών συναρτήσεων για την παράγωγη κλάση μπορούμε να χρησιμοποιήσουμε συναρτήσεις που γράψαμε για τη βασική κλάση. Ας δούμε δύο παραδείγματα.

Επιφορτώνουμε τον "==" για τη *DateTime* ως εξής:

```
bool operator==( const DateTime& lhs, const DateTime& rhs )
{
   return ( static_cast<Date>(lhs) == static_cast<Date>(rhs) &&
            lhs.getHour() == rhs.getHour() &&
            lhs.getMin() == rhs.getMin() &&
            lhs.getSec() == rhs.getSec() );
} // operator==( const DateTime
```

και τον "<<":

```
ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return tout << static_cast<Date>( rhs ) << ' '
               << rhs.getHour() << ':' << rhs.getMin() << ':'
               << rhs.getSec();
} // operator<<( ostream& tout, const DateTime
```

23.7 Κλάση Εξαιρέσεων Παράγωγης Κλάσης

Θα τηρούμε τον εξής κανόνα:

- ***Η κλάση εξαιρέσεων της παράγωγης κλάσης είναι παράγωγη της κλάσης εξαιρέσεων της βασικής κλάσης.***

Θα έχουμε, για παράδειγμα:

```
struct DateTimeXptn :  public DateXptn
{
   enum { hourRange = 50, minRange, secRange  };
   DateTime errDateTimeVal;
   DateTimeXptn( const char* mn, int ec,
                 int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : DateXptn( mn, ec, ev1, ev2, ev3 ) {  }
   DateTimeXptn( const char* mn, int ec, int ev1, const DateTime& edt )
      : DateXptn( mn, ec, ev1, edt )
   {  errDateTimeVal = edt;  }
}; // DateTimeXptn
```

Πρόσεξε:

- Το "`hourRange = 50`": με αυτόν τον τρόπο επιδιώκουμε να διαχωρίσουμε τις περιοχές κωδικών σφάλματος των *DateXptn* (0 .. *DateXptn*::*cannotWrite*) και *DateTimeXptn* (50.. *DateTimeXptn*::*secRange*).
- Τη σύλληψη των εξαιρέσεων· αν κάπου πιάνεις εξαιρέσεις και των δύο τύπων θα πρέπει να βάλεις:

  ```
  catch( DateTimeXptn& x )
  { . . . }
  catch( DateXptn& x )
  { . . . }
  ```

  Αν βάλεις τις **catch** με την αντίστροφη σειρά η "`catch(DateXptn& x)`" δεν θα αφήνει να περάσει οποιαδήποτε εξαίρεση στην "`catch(DateTimeXptn& x)`", αφού κάθε `DateTimeXptn` είναι και `DateXptn`.

Και μια παρατήρηση για το συγκεκριμένο παράδειγμα: Ένα αντικείμενο κλάσης *DateTimeXptn* έχει μέλος κλάσης *DateTime* αλλά έχει κληρονομήσει και μέλος κλάσης *Date* (*errDateVal*). Στο κληρονομημένο μέλος δίνουμε τιμή με τη λίστα εκκίνησης του δεύτερου

δημιουργού, από το υποαντικείμενο κλάσης *Date* του *edt*, με τον σχετικό τεμαχισμό. Ολόκληρο το *edt* αντιγράφεται στο νέο μέλος *errDateTimeVal*.

23.8 "private" ή "protected"

Λέγαμε στην §23.3 ότι «*ο δημιουργός της DateTime δεν έχει πρόσβαση στα dYear, dMonth, dDay του υποαντικειμένου Date! Για όλα φταίει εκείνο το "*`private`*:" που έχουμε στη Date*». Αυτό βέβαια ισχύει και για οποιαδήποτε μέθοδο της *DateTime*.

Παρ' ολα αυτά, όπως φάνηκε από αυτά που είδαμε στις προηγούμενες παραγράφους:

- Μπορούμε να αλλάζουμε τις τιμές των μελών του υποαντικειμένου της βασικής με κάποιον δημιουργό ή άλλες μεθόδους (π.χ. *swap()*) και –αν χρειαστεί– με κάποια "*set*".
- Μπορούμε να παίρνουμε τις τιμές των μελών του υποαντικειμένου της βασικής με τις "*get*".

Σε πολλές περιπτώσεις πάντως, όταν έχουμε πολύπλοκους υπολογισμούς, τα παραπάνω δεν είναι ικανοποιητικά. Εκτός από αυτό, συχνά στις περιοχές `private` υπάρχουν «κρυφές» μέθοδοι ή βοηθητικές συναρτήσεις που μας είναι χρήσιμες για την υλοποίηση των μεθόδων της παράγωγης κλάσης.

Η C++ μας δίνει λύση στα προβλήματα αυτά με μια τρίτη κατηγορία δηλώσεων, τις δηλώσεις "`protected`".

- ***Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της.***

Σημείωση: ▶

Δες μια άλλη κάπως ακριβέστερη διατύπωση: *Ό,τι υπάρχει στην περιοχή "*`protected`*" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης*. Διάβασε την επόμενη υποπαράγραφο.◀

Ας δούμε ένα παράδειγμα· αλλάζουμε τη *Date* ως εξής:

```
class Date
{
// . . .
public:
// . . .
protected:
   unsigned int  dYear;
   unsigned char dMonth;
   unsigned char dDay;

   static unsigned int lastDay( int y, int m );
private:
   static bool leapYear( int y );
}; // Date
```

Τώρα στον δημιουργό της *DateTime* μπορείς να γράψεις, αφού βάλεις τους ελέγχους που ξέρουμε:

```
   if ( yp <= 0 ) throw . . .
   dYear = yp;
   if ( mp <= 0 || 12 < mp ) throw . . .
   dMonth = mp;
   if ( dp <= 0 || lastDay(dYear, dMonth) < dp )
      throw DateXptn( "Date", DateXptn::dayErr, yp, mp, dp );
   dDay = dp;
```

αντί για

```
   :  Date( yp, mp, dp )
```

ή

```
   *(static_cast<Date*>(this)) = Date( yp, mp, dp );
```

Αλλά προσοχή: αυτό δεν σημαίνει ότι πρέπει να γράψεις έτσι τον δημιουργό. Όπως βλέπεις, ξαναγράφουμε και εδώ τους ελέγχους που έχουμε γράψει στον δημιουργό και στις "*set*" της *Date*. Οι δύο τρόποι, που είδαμε πιο πριν είναι προτιμότεροι διότι βασίζονται στον δημιουργό της *Date* που έχει τους ελέγχους.

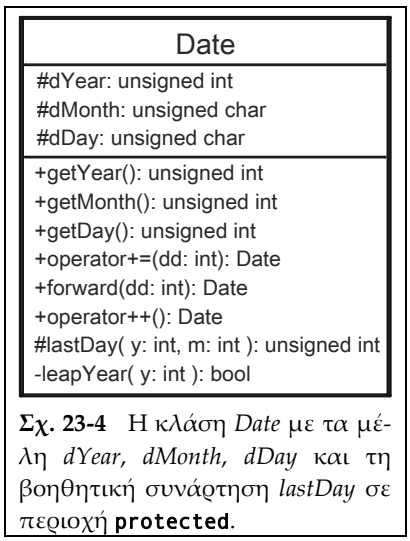
Αν πάμε όμως στη *forword()* τα πράγματα αλλάζουν: οι

```
dYear = currD.tm_year+1900;
dMonth = currD.tm_mon+1;
dDay = currD.tm_mday;
```

είναι προτιμότερες από την

```
*(static_cast<Date*>(this)) = Date( currD.tm_year+1900,
                                    currD.tm_mon+1,
                                    currD.tm_mday );
```

διότι ο δημιουργός που καλείται θα κάνει ελέγχους που είναι άχρηστοι.



Σχ. 23-4 Η κλάση *Date* με τα μέλη *dYear*, *dMonth*, *dDay* και τη βοηθητική συνάρτηση *lastDay* σε περιοχή `protected`.

Όπως λέγαμε και στην §20.8.1, στα διαγράμματα της UML, ό,τι δηλώνεται σε περιοχή "`protected`" σημειώνεται με "`#`". Στο Σχ. 23-4 βλέπεις διαγραμματική παράσταση της *Date*, όπως την αλλάξαμε παραπάνω.

Πολλοί πεπειραμένοι προγραμματιστές βάζουν πάντοτε τα μέλη σε περιοχή "`private`" ενώ σε περιοχή "`protected`" μπορεί να βάλουν βοηθητικές συναρτήσεις ή μεθόδους.

Άλλοι, χωρίς συζήτηση, αντικαθιστούν το "`private`" με το "`protected`" σε κάθε κλάση που πρόκειται να κληρονομηθεί. Αν θέλεις να κάνεις και εσύ το ίδιο πάρε υπόψη σου τα εξής:

- ***Αν χρησιμοποιείς την αλλαγή περιορισμού πρόσβασης από "`private`" σε "`protected`" –για να μπορείς να αλλάζεις τις τιμές των μελών του υποαντικειμένου της βασικής κλάσης– θα πρέπει να βάζεις και τους απαραίτητους ελέγχους κατά περίπτωση.***

Φυσικά, με αυτόν τον τρόπο, πληθαίνουν τα σημεία των ελέγχων πράγμα που κάνει πιο πολύπλοκη τη κάθε απόπειρα ενημέρωσης/τροποποίησης της βασικής κλάσης. Για τον λόγο αυτόν, ακόμη και με τη αλλαγή σε "`protected`", καλό είναι

- ***Να προσπαθείς να ελαχιστοποιείς την κατ' ευθείαν πρόσβαση στα μέλη του υποαντικειμένου γράφοντας τις μεθόδους της παράγωγης κλάσης με χρήση των μεθόδων και των δημιουργών της βασικής.***

23.8.1 * "`protected`": Ψιλά Γράμματα

Επανερχόμαστε τώρα στον κανόνα-ορισμό της προηγούμενης παραγράφου: «*Ό,τι βάλουμε στην περιοχή "`protected`" μιας κλάσης είναι ανοικτό στις παράγωγες κλάσεις της*» και στη διευκρινιστική σημείωση «*Ό,τι υπάρχει στην περιοχή "`protected`" του υποαντικειμένου της βασικής κλάσης είναι ανοικτό στις μεθόδους του αντικειμένου της παράγωγης.*» Ας πούμε ότι έχουμε:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   void display( ostream& tout ) const {  tout << mb;  }
protected:
   int mb;
}; // B
```

```
class D2: public B
{
public:
   D2( int bp=2, int d2p=0 ): B(bp)  {  md2 = d2p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md2;  }
private:
   int md2;
}; // D2
```

Θέλουμε να εφοδιάσουμε τη *D2* με μια μέθοδο:

```
   void xchangeB( B& b );
```

που θα ανταλλάσσει τις τιμές των μελών *mb* του αντικειμένου και του *b*. Αν

```
   B b;
   D2 d2a( 2, 11 );
```

τότε οι:

```
   d2a.xchangeB( b );
   d2a.display( cout );  cout << endl;
   b.display( cout );  cout << endl;
```

θα πρέπει να δώσουν:

```
0 11
2
```

Θέλουμε να υλοποιήσουμε τη μέθοδο ως εξής:

```
   void xchangeB( B& b )  {  std::swap( b.mb, mb );  }
```

αλλά ο μεταγλωττιστής[8] έχει αντιρρήσεις: «`Error E2247 testDerived.cpp 31: 'B::mb' is not accessible in function D2::xchangeB(B &)`».

Εδώ τι έχουμε;

- Στη βασική κλάση (*B*) έχουμε δηλώσει "`protected: int mb`".
- Στην παράγωγη κλάση (*D1*), μια μέθοδος προσπαθεί να διαχειρισθεί το μέλος *mb* ενός αντικειμένου *b* κλάσης *B*.

Ο μεταγλωττιστής μας λέει ότι το "`B::mb`" δεν είναι προσβάσιμο από τη συνάρτηση-μέλος "`D2::xchangeB(B &)`".

Αν βάλουμε:

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

ο μεταγλωττιστής δεν έχει πρόβλημα με τη μέθοδο. Αν έχουμε δηλώσει

```
   D2 d2a( 2, 11 ), d2b( 3, 13 );
```

οι

```
   d2a.xchangeB( d2b );
   d2a.display( cout );  cout << endl;
   d2b.display( cout );  cout << endl;
```

θα δώσουν:

```
3 11
2 13
```

Τώρα όμως δεν είναι δεκτή η "`d2a.xchangeB( b )`".

Αν ορίσουμε

```
class DD2: public D2
{
public:
   DD2( int bp=1, int d2p=0, int dd2p=0 )
      : D2(bp, d2p)  {  mdd2 = dd2p;  }
   void display( ostream& tout ) const
   {  D2::display( tout );  tout << ' ' << mdd2;  }
```

[8] Borland BC++, v.5.5.

```
private:
   int mdd2;
}; // DD2
```

και δηλώσουμε:

```
   DD2 dd2( 5, 17, 23 );
```

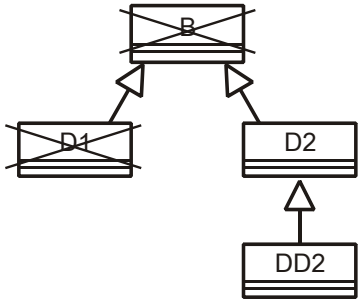
η

```
   d2a.xchangeB( dd2 );
```

είναι δεκτή και δουλεύει.

Αν όμως ορίσεις

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 )
      : B(bp)  {  md1 = d1p;  }
   void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
private:
   int md1;
}; // D1
```



Σχ. 23-5 Οι μέθοδοι της *D2* «βλέπουν» τα "`protected`" μέλη των υποαντικειμένων *B* των αντικειμενων της *D2* και της *DD2*· όχι της *B* και της *D1*.

Αλλάζουμε τώρα την *xchangeB()* σε

```
   void xchangeB( D1& b )  {  std::swap( b.mb, mb );  }
```

Και πάλι το ίδιο πρόβλημα: δεν επιτρέπεται πρόσβαση προς το "`b.mb`"!

Διατυπώνουμε λοιπόν τον κανόνα μας ως εξής (Σχ. 23-5):

- ***Αν έχουμε μια (βασική) κλάση B στην οποίαν υπάρχει περιοχή "`protected`", μια κλάση D, παράγωγη (αμέσως ή εμμέσως) της B και μια μέθοδο f της D, η f έχει πρόσβαση στα μέλη των περιοχών "`protected`" των υποαντικειμένων κλάσης B των αντικειμένων κλάσης D και των παραγώγων της D.***

23.9 Εικονικές Μέθοδοι

Στην §23.5 λέγανε: «*Αν στην παράγωγη κλάση ορίσουμε μέθοδο με το όνομα μιας μεθόδου της βασικής τα αντικείμενα της παράγωγης βλέπουν τη μέθοδο που ορίστηκε στην παράγωγη κλάση.*» Είναι όμως πάντα έτσι; Δες ένα πρόβλημα: Δηλώνουμε

```
   Date* pind[2];
```

και παίρνουμε μνήμη:

```
   pind[0] = new Date( 2002, 2, 2 );
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

Το δεύτερο είναι σωστό; Είναι και παραείναι! Αφού «*Οπουδήποτε … μπορούμε να βάλουμε αντικείμενο της βασικής κλάσης B, την οποίαν η D κληρονομεί, μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης D*» όλα είναι εντάξει. Για δες όμως τη συνέχεια:

```
   pind[0]->forward( 10 );
   pind[1]->forward( 10 );
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
   ++(*pind[0]);
   ++(*pind[1]);
   cout << pind[0]->getDay() << endl;
   cout << pind[1]->getDay() << endl;
```

Αποτέλεσμα:

```
12
13
13
14
```

Δηλαδή; Και για το δεύτερο αντικείμενο χρησιμοποίησε τη *forward()* και τον "++" της βασικής. Η ζημιά έγινε από τον μεταγλωττιστή· όταν έκανε τη μεταγλώττιση σημείωσε ότι τα βέλη του pind δείχνουν αντικείμενα κλάσης *Date* και αυτό ήταν... Όταν έρχεται η ώρα του υπολογισμού χρησιμοποιεί τις μεθόδους της *Date*.

Τώρα σου έρχεται αγανάκτηση: «Ε, δεν θα χρησιμοποιούμε βέλη και τελειώσαμε!» Εύκολα το λες, δύσκολα το κάνεις. Ας πούμε ότι έχουμε:

```
   DateTime dt( 2003, 3, 3, 3, 33, 33 );
```

και τη συνάρτηση:

```
void plus10( Date& d )
{
   d.forward( 10 );
} // plus10
```

Οι εντολές

```
   cout << dt << endl;
   plus10( dt );
   cout << dt << endl;
```

θα δώσουν:

```
3.3.2003 3:33:33
13.3.2003 3:33:33
```

Δηλαδή, κλήθηκε η *forward()* της *Date* και όχι της *DateTime*. Γιατί; Αν το έχεις ξεχάσει, να υπενθυμίσουμε ότι

- οι παράμετροι αναφοράς είναι «μεταμφιεσμένα» βέλη και ότι
- όταν θέλουμε να περάσουμε σε συνάρτηση αντικείμενο θα χρησιμοποιήσουμε παράμετρο αναφοράς (με `const` ή χωρίς), διότι τα αντικείμενα είναι συνήθως μεγάλα.

Καταλαβαίνεις λοιπόν ότι δεν μπορείς να πεις τόσο εύκολα ότι δεν θα χρησιμοποιήσεις βέλη.

Πώς διορθώνεται αυτό; Ως εξής:

♦ ***Όποιες μεθόδους ξαναορίζουμε στην παράγωγη κλάση τις δηλώνουμε στη βασική κλάση ως "`virtual`" (εικονικές).***

```
class Date
{
// . . .
public:
   virtual const Date& operator+=( long int dd );
   virtual const Date& forward( long int dd );
   virtual Date& operator++();
// . . .
}; // Date
```

Μετά από αυτή τη διόρθωση οι εντολές του πρώτου παραδείγματος θα δώσουν:

```
12
3
13
3
```

ενώ από το δεύτερο θα πάρουμε:

```
3.3.2003 3:33:33
3.3.2003 3:33:43
```

Τι κάνει αυτή η «μαγική» λέξη;

- Ο χαρακτηρισμός "`virtual`" (εικονικός) μπαίνει σε μεθόδους της *βασικής* κλάσης όταν στην παράγωγη κλάση υπάρχει μέθοδος με το ίδιο όνομα (πολυμορφική). Στην περίπτωσή μας βάλαμε "`virtual`" τις `operator+=()`, *forward()*, `operator++()` της *Date* διότι αυτές ορίζονται ξανά και στην παράγωγή της *DateTime*.

- Το αποτέλεσμά του είναι το εξής: προειδοποιεί τον μεταγλωττιστή να μην βιαστεί να «δέσει» τη συγκεκριμένη μέθοδο με τα βέλη προς αντικείμενα της βασικής αλλά να αναβάλει την απόφασή του μέχρι τη στιγμή της εκτέλεσης της εντολής ώστε να μπορεί να «δει» *την κλάση του αντικειμένου που δείχνει το βέλος εκείνη τη στιγμή*.

Να βάλουμε "`virtual`" και στις αντίστοιχες μεθόδους της παράγωγης κλάσης; Πρόσεξε: Αν κατάλαβες τι ακριβώς συμβαίνει με το "`virtual`" θα έχεις καταλάβει ότι και αυτές είναι "`virtual`" είτε βάλουμε είτε δεν βάλουμε.[9] Καλύτερα λοιπόν να γράψουμε το "`virtual`" και στην παράγωγη κλάση για λόγους τεκμηρίωσης. Γιατί να μην δούμε την αλήθεια κατάματα;...

23.10 * Υλοποίηση Εικονικών Συναρτήσεων

Θα πούμε τώρα δυο λόγια για την υλοποίηση των εικονικών μεθόδων, παρ' όλο που αυτό είναι έξω από το αντικείμενο αυτού του βιβλίου.

Με τις κλάσεις του παραδείγματος της §23.3 (κάπως τροποποιημένες)

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual void display( ostream& tout ) const {  tout << mb;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B

class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void display( ostream& tout ) const
   {  B::display( tout );  tout << ' ' << md1;  }
   virtual void g( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

κάνουμε το εξής πείραμα: Δηλώνουμε δύο αντικείμενα:

```
   B   bo;
   D1  d1o;
```

και ζητούμε:

```
   cout << "sizeof_bo: " << sizeof(bo) << endl;
   cout << "sizeof_d1o: " << sizeof(d1o) << endl;
```

Αποτέλεσμα:[10]

```
sizeof_bo: 8
sizeof_d1o: 12
```

Αλλά σε ένα αντικείμενο κλάσης *B* έχουμε ένα μέλος τύπου `int` ενώ τα αντικείμενα κλάσης *D1* έχουν δύο τέτοια μέλη. Ο μεταγλωττιστής που χρησιμοποιούμε παριστάνει τις τιμές τύπου `int` με 4 ψηφιολέξεις. Και στις δύο περιπτώσεις «περισσεύουν» 4 ψηφιολέξεις. Με αυτά υλοποιείται μια «κρυφή» μεταβλητή-βέλος, ας την πούμε *vfptr*, που δείχνει έναν πίνακα με τις εικονικές μεθόδους (για την ακρίβεια τις διευθύνσεις των εικονικών μεθό-

[9] «*Once virtual, always virtual!*»

[10] g++ (Dev-C++).

δων) που χρησιμοποιούν τα αντικείμενα της κλάσης και ονομάζεται **πίνακας εικονικών μεθόδων / συναρτήσεων** (virtual method / function table) ή απλώς **vtable**.

Κατά τη διάρκεια της μεταγλώττισης:

- Στον vtable της *B* θα εισαχθούν οι *B*::*f* και *B*::*display*.
- Στον vtable της *D1* θα εισαχθούν οι *B*::*f* (που κληρονομείται από τη *B*), *D1*::*display* και *D1*::*g*.
- Κάθε αντικείμενο κλάσης *B*, όπως το *bo*, εφοδιάζεται με το «κρυφό» βέλος *vfptr* και στους δημιουργούς της κλάσης μπαίνουν εντολές που το βάζουν να δείχνει τον vtable της *B*.
- Τα αντίστοιχα γίνονται και με τα αντικείμενα κλάσης *D1*.

Κατά τη διάρκεια της εκτέλεσης αν ζητηθεί η εκτέλεση κάποιας εικονικής μεθόδου ενός αντικειμένου αναζητείται η διεύθυνσή της στον πίνακα που δείχνει το *vfptr* και εκτελείται η σωστή μέθοδος.

23.10.1 "virtual" και Τεμαχισμός

Είναι δυνατόν να βάλουμε το "`virtual`" και να μην έχουμε τη σωστή συμπεριφορά του αντικειμένου; Και βέβαια, αν δεν το αφήνουμε να λειτουργήσει. Δες ένα

Παράδειγμα

Χρειαζόμαστε μια συνάρτηση, ας την πούμε *plus10Local()*, που θα τροφοδοτείται με ένα αντικείμενο *Date* ή *DateTime* –όπως η *plus10()*– και θα προχωρεί την τιμή του κατά 10 ημέρες ή *sec* αντιστοίχως αλλά μόνο για τις επεξεργασίες που γίνονται μέσα στη συνάρτηση· η αλλαγή δεν θα βγαίνει προς τη συνάρτηση που θα καλεί την *plus10Local()*.

Θα έλεγε κανείς –με αρκετή επιπολαιότητα– ότι εδώ μας δίνεται η ευκαιρία να «γλυτώσουμε» από τα βέλη και να γράψουμε:

```
void plus10Local( Date d )
{
   d.forward( 10 );
// . . .
} // plus10Local
```

Έτσι κάναμε ένα σοβαρό λάθος: όταν εκτελείται η "`plus10Local(dt)`" καλείται ο δημιουργός αντιγραφής της *Date* για να αντιγράψει το *dt* στο *d*. Το *dt*; Όχι! Το υποαντικείμενο κλάσης *Date* του *dt*. Έχουμε δηλαδή *τεμαχισμό* και το "`virtual`" δεν έχει ευκαιρία να δουλέψει...

Δυστυχώς, το λάθος αυτό γίνεται με τρόπο που είναι τελείως «νόμιμος» για τον μεταγλωττιστή και έτσι δεν θα μας δώσει κάποιο μήνυμα.

Καταλαβαίνεις τώρα ότι η συνήθεια να περνούμε τα αντικείμενα με παραμέτρους αναφοράς δεν έχει να κάνει μόνο με την αποφυγή μιας αντιγραφής.[11]

Παρατήρηση: ▶

Μια και συζητούμε για αντιγραφές, δες ένα πρόβλημα που προκύπτει: Έστω ότι βάζουμε παράμετρο "`const Date& d`". Τώρα η "`d.forward(10)`" είναι παράνομη λόγω του "`const`". Για να κάνουμε τη δουλειά μας θα πρέπει να αντιγράψουμε το *d* σε ένα αντικείμενο, τοπικό στην *plus10Local*. Αλλά, τι τύπου θα είναι αυτό το τοπικό αντικείμενο; Σε λίγο θα το μάθουμε και αυτό...◀

[11] Ο κανόνας OBJ33 του (CERT 2009) λέει: *Μην τεμαχίζεις πολυμορφικά αντικείμενα* (*Do not slice polymorphic objects*).

23.10.2 Κάποια Σχόλια...

- Ο όρος **virtual** αποδόθηκε στα ελληνικά ως «*εικονική*»· θα βρεις επίσης και το όρο «*νοητή*». Και οι δύο μεταφράσεις είναι εξ ίσου παραπλανητικές. Όπως είδαμε μέχρι τώρα, οι μέθοδοι που δηλώνονται ως "`virtual`" είναι πραγματικότατες. Μια (περιφραστική) μετάφραση «*δυναμικώς επιλεγόμενη*» θα ήταν πιο κοντά στην πραγματικότητα. Πάντως στη συνέχεια θα δούμε μεθόδους που είναι γνησίως εικονικές.
- Γιατί να μην είναι όλες οι μέθοδοι "**virtual**" να τελειώνουμε; Διότι ο μηχανισμός επιλογής της σωστής μεθόδου χρειάζεται χρόνο εκτέλεσης. Γιατί λοιπόν να φορτώνουμε το πρόγραμμά μας με άχρηστες δουλειές για μεθόδους που δεν πρόκειται να αλλάξουμε, π.χ. για τις "*get*" και "*set*";
- Το πρόβλημα που λύνεται με τις εικονικές μεθόδους υπάρχει και στις άλλες αντικειμενοστρεφείς γλώσσες και οι λύση είναι παρόμοια: «**καθυστερημένη**» **επιλογή** (**πρόσδεση**) (late ή deferred binding) μεθόδου. Οι «συγγενεις» γλώσσες –Java και C#– μπορεί να μην χρησιμοποιούν (φανερά) βέλη αλλά χρησιμοποιούν «κρυμμένα» (αναφορές). Έτσι
  - Στη Java κάθε μέθοδος είναι κατ' αρχήν "**virtual**" και ο προγραμματιστής θα πρέπει *να δηλώσει ποιες δεν θέλει* να είναι.
  - Η C# ακολουθεί τη C++: κάθε μέθοδος, κατ' αρχήν, *δεν είναι* "**virtual**" και ο προγραμματιστής θα πρέπει να δηλώσει ποιες θέλει να είναι.

23.11 Εικονικός Καταστροφέας

Επιστρέφουμε στο παράδειγμα της §23.3, όπως το εμπλουτίσαμε με τους καταστροφείς στην §23.4. Δίνουμε τις εντολές:

```
   B* pd( new D );
   delete pd;
```

και παίρνουμε:

```
B object created
A object created
C object created
D object created
destroying B object
```

Το ενδιαφέρον βρίσκεται στην τελευταία γραμμή, «`destroying B object`», που δείχνει ότι καλείται ο καταστροφέας της βασικής κλάσης για να καταστρέψει ένα αντικείμενο της παράγωγης κλάσης, το `*pd`. Πώς έγινε αυτό; Η απόφαση για το είδος του καταστροφέα πάρθηκε πολύ νωρίς, στη μεταγλωττιση.

Πώς διορθώνεται αυτό; Όπως είδαμε στην προηγούμενη παράγραφο:

♦ ***Ο καταστροφέας της βασικής κλάσης πρέπει να δηλώνεται "`virtual`".***

Πράγματι, αν ορίσουμε:

```
   virtual ~B() {  cout << "destroying B object" << endl;  }
```

θα πάρουμε:

```
B object created
A object created
C object created
D object created
destroying D object
destroying C object
destroying A object
destroying B object
```

Στην §21.5 λέγαμε «*Πολλοί προγραμματιστές συνηθίζουν να βάζουν έναν «μηδενικό» καταστροφέα "`~K() {}`" στις κλάσεις που δεν χρειάζονται. Αργότερα θα δούμε ότι σε*

μερικές περιπτώσεις αυτό είναι απαραίτητο.» Τώρα βλέπεις ότι αυτό είναι απαραίτητο στις κλάσεις που πρόκειται να κληρονομηθούν· και μάλιστα θα πρέπει να ορίζεις:

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

Παράδειγμα

Θα δούμε στη συνέχεια ότι η C++ έχει μια βασική κλάση εξαιρέσεων, την `std::exception`, από την οποία παράγονται όλες οι κλάσεις εξαιρέσεων που χρησιμοποιούν οι συναρτήσεις της βιβλιοθήκης της. Αυτή η κλάση είναι περίπου έτσι:

```
struct exception
{
   exception() throw() { }
   virtual ~exception() throw();
   virtual const char* what() const throw();
}; // exception
```

Όπως βλέπεις, ο καταστροφέας δεν ορίζεται αλλά έχει στη δήλωσή του το "`virtual`".

Έτσι, μπορούμε να βάλουμε στις δικές μας κλάσεις:

```
struct DateXptn :  public exception
{
// . . .
   virtual ~DateXptn() throw() { };
}; // DateXptn
```

και

```
struct DateTimeXptn :  public DateXptn
{
// . . .
   virtual ~DateTimeXptn() throw() { };
}; // DateTimeXptn
```

Αν θελήσεις να δοκιμάσεις τα παραπάνω θα πρέπει να βάλεις στο πρόγραμμά σου "`#include <exception>`".

23.12 Περί Πολυμορφισμού

Αφού είδαμε διάφορες πλευρές –τεχνικές κυρίως– του πολυμορφισμού ας ολοκληρώσουμε –προς το παρόν– αυτό το θέμα, κυρίως για να ξεδιαλύνουμε μερικές παρεξηγήσεις της μορφής «πολυμορφισμός είναι επιφόρτωση ή υπερίσχυση ή μέθοδοι **virtual**».

Γενικώς, μπορούμε να ορίσουμε τον **πολυμορφισμό** (polymorphism) ως τη δυνατότητα να χειριζόμαστε (εν μέρει συνήθως) έναν τύπο *A* όπως τον τύπο *B*.

Στα πλαίσια του αντικειμενοστρεφούς προγραμματισμού αυτό εξειδικεύεται ως: η δυνατότητα αντικειμένων *διαφορετικών κλάσεων* να αποκρίνονται στο ίδιο μηνυμα με αποκρίσεις –όχι ταυτόσημες, αλλά– που εξαρτώνται από την κλάση του κάθε αντικειμένου.

Αυτά που μάθαμε μέχρι τώρα αφορούν την υλοποίηση στη C++ του λεγόμενου **πολυμορφισμού υποκλάσεων** (subclass ή subtype). Δηλαδή: αντικείμενα διαφορετικών τύπων που ανήκουν στην ίδια ιεραρχία κλάσεων μπορεί να δεχθούν κλήση μεθόδου ή τελεστή με το ίδιο όνομα (π.χ. `d.forward(10)` ή `d += 10`) και αποκρίνονται με τρόπο που εξαρτάται από την κλάση του αντικειμένου (π.χ. αν το *d* είναι κλάσης *Date* προχωρεί την ημερομηνία κατά 10 ημέρες ενώ αν είναι κλάσης *DateTime* προχωρεί τον χρόνο κατά 10 *sec*).

Πώς γίνονται αυτά; Η παράγωγη κλάση (υποκλάση) κληρονομεί από τη βασική (υπερκλάση) τη μέθοδο (τελεστή) που μας ενδιαφέρει (π.χ.: `const Date& forward(long int dd)`). Στην παράγωγη κλάση *επιφορτώνουμε*(;) μια μέθοδο με το ίδιο όνομα και *παρόμοια επικεφαλίδα* (π.χ.: `const DateTime& forward( long int dd)`). Η μέθοδος αυτή

υπερισχύει[12] αυτής που κληρονομήθηκε και *σχεδόν* κάθε φορά που στέλνουμε το σχετικό μήνυμα σε αντικείμενο της παράγωγης αυτό αποκρίνεται με την υπερισχύουσα μέθοδο. Για να γίνει εκείνο το «*σχεδόν* κάθε φορά» απλώς «κάθε φορά» δηλώνουμε τις μεθόδους μας "`virtual`".

Στην παραπάνω ιστορία έχουμε αφήσει κάποια πράγματα που χρειάζονται διευκρίνιση: ένα ερωτηματικό(;) και μια «*παρόμοια επικεφαλίδα*».

- Πρώτα το «*επιφορτώνουμε*(;)». ΔΕΝ επιφορτώνουμε, τουλάχιστον με την έννοια που ξέρουμε μέχρι τώρα. Στην επιφόρτωση συναρτήσεων μάθαμε ότι οι διάφορες συναρτήσεις με το ίδιο όνομα θα πρέπει να έχουν *διαφορετικές υπογραφές*. Δηλαδή θα πρέπει να έχουν *διαφορά* στις παραμέτρους (ώστε να μπορεί ο μεταγλωττιστής να επιλέξει). Οι πολυμορφικές μέθοδοι *πρέπει να έχουν τις ίδιες ακριβώς παραμέτρους*.
- Πόσο «*παρόμοια επικεφαλίδα*»; Αφού έχουμε ίδιο όνομα και ίδιες παραμέτρους, το «*παρόμοια*» πάει στον τύπο (επιστρεφόμενης τιμής) της συνάρτησης. Ο τύπος μπορεί να είναι ο ίδιος ή –αν δεν είναι– θα πρέπει να είναι *μετατρέψιμος* τύπος βέλους ή αναφοράς. Όπως θα δεις παρακάτω, ο τύπος βέλους της βασικής κλάσης (π.χ. `Date*` ή `Date&`) μπορεί να μετατραπεί σε τύπο βέλους προς την παράγωγη (π.χ. `DateTime*` ή `DateTime&`)· ο τύπος της μεθόδου της παράγωγης κλάσης είναι βέλος παράγωγης κλάσης.[13]

Για παράδειγμα, μπορείς να δεις αυτά τα πράγματα στη μέθοδο *forward()* ή στον "`+=`" των *Date* και *DateTime*:

```
const Date& Date::forward( long int dd )
const DateTime& DateTime::forward( long int ds )
```

ή

```
const Date& Date::operator+=( long int dd )
const DateTime& DateTime::operator+=( long int ds )
```

Και στις δύο περιπτώσεις μπορούσαμε να τις κάνουμε "`virtual`" και τις κάναμε.

Αυτός ο –δυναμικός– πολυμορφισμός ονομάζεται **πολυμορφισμός χρόνου εκτέλεσης** (run-time).

Παρατήρηση: ▶

Αν στη βασική κλάση, μια μέθοδος "`virtual`" έχει προδιαγραφή εξαιρέσεων, οι υπερισχύουσες μέθοδοι στις παράγωγες κλάσεις θα πρέπει να έχουν την ίδια ακριβώς προδιαγραφή εξαιρέσεων. Στο παράδειγμα της §23.11 μας δίνεται η βασική κλάση με υπογραφή καταστροφέα:

```
virtual ~exception() throw();
```

Στην παράγωγη κλάση (*DateXptn*) θα πρέπει να βάλουμε:

```
virtual ~DateTimeXptn() throw() { };
```

Αν παραλείψεις το "`throw()`" ο μεταγλωττιστής θα σου βγάλει λάθος.◀

23.12.1 Πολυμορφισμός Χρόνου Μεταγλώττισης(;)

Ας πούμε ότι μας ενδιαφέρει να κάνουμε πολυμορφικό τον μεταθεματικό "++". Όπως έχουμε ήδη πει, αυτός *δεν μπορεί να επιστρέψει βέλος ή αναφορά* και θα είναι:

```
Date Date::operator++( int )
DateTime DateTime::operator++( int )
```

Μπορούμε να το αφήσουμε έτσι· στην παράγωγη θα έχουμε απόκρυψη του τελεστή της βασικής. Αλλά

---

[12] Από την στιγμή που έχουμε *υπερίσχυση* παύουμε να μιλούμε για *επιφόρτωση*. Κρατούμε αυτόν τον όρο για την περίπτωση που η επικεφαλίδα δεν είναι *παρόμοια*. Το συζητούμε στη συνέχεια.

[13] Αυτό ονομάζεται *συμμεταβλητότητα* (covariance).

- Θα έχουμε διαφορετική συμπεριφορά των αντικειμένων της παράγωγης όταν τα χειριζόμαστε με βέλη και αναφορές της βασικής.
- Δεν μπορούμε να βάλουμε το "`virtual`" για να λύσουμε το πρόβλημα.

Αυτός ο πολυμορφισμός, που μας δίνει πιο γρήγορο πρόγραμμα, ονομάζεται πολυμορφισμός **χρόνου μεταγλώττισης** (compile time) σε αντιδιαστολή με τον «πλήρη» πολυμορφισμό που είναι **χρόνου εκτέλεσης** (run-time). Αλλά όπως καταλαβαίνεις μπορεί να δημιουργήσει πολύ σοβαρά προβλήματα στο πρόγραμμά σου· μην τον χρησιμοποιείς! Γενικώς, όπως μας συμβουλεύει το (CERT 2009):[14]

♦ ***Μην «κρύβεις» κληρονομούμενες μεθόδους που δεν είναι "virtual".***

23.12.2 * Υπερίσχυση ή Επιφόρτωση

Μέχρι τώρα ξέραμε την *επιφόρτωση* (overloading) συναρτήσεων· στο κεφάλαιο αυτό μάθαμε την *υπερίσχυση* (overriding) που είναι κάτι τελείως διαφορετικό. Αν τις μπερδέψεις μπορεί να έχεις προβλήματα, όπως φαίνεται στο παρακάτω παράδειγμα.

Επιστρέφουμε ξανά στο πρόγραμμα της §23.3 και κάνουμε μερικές αλλαγές, στη *B*:

```
class B
{
public:
   B( int bp=0 )  {  mb = bp;  }
   virtual int f() const
   {  cout << "B::f()" << endl;  return mb+1;  }
protected:
   int mb;
}; // B
```

και στη *D1*:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
private:
   int md1;
}; // D1
```

Η *f()* της *D1 υπερισχύει* –στα αντικείμενα κλάσης *D1*– της *f()* που ορίσαμε στη *B*. Αν βάλουμε στη `main`

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   B b( d );
   cout << b.f() << endl;
```

θα πάρουμε:

```
D1::f()
3
B::f()
2
```

Ας πούμε τώρα ότι αλλάζουμε τη *D1* ως εξής:

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
```

[14] Η σύσταση *OBJ02* λέει: «*Do not hide inherited non-virtual member functions*».

```
private:
   int md1;
}; // D1
```

και στη `main` ζητούμε:

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Πρόσεξε τι έχουμε εδώ: Στη βασική κλάση υπάρχει μια μέθοδος

```
   virtual int f() const;
```

και στην παράγωγη μια:

```
   virtual void f( int& aV, int& aDV ) const;
```

Τι σχέση έχουν οι δύο μέθοδοι; Καμιά, απλώς έχουν το ίδιο όνομα. Δηλαδή, στην παράγωγη κλάση, που κληρονομεί την *f()* της *B*, έχουμε *επιφόρτωση*, Τα "`virtual`" δεν παίζουν οποιοδήποτε ρόλο, τουλάχιστον σε ό,τι βλέπουμε!

Δυστυχώς, ο μεταγλωττιστής θα βγάλει πρόβλημα στην:

```
   cout << d.f() << endl;
```

(«`Error E2193 tO.cpp 63: Too few parameters in call to 'D1::f(int &,int &) const' in function main()`»[15]) που, με απλά λόγια, οφείλεται στο εξής: ο μεταγλωττιστής θα αναζητήσει μέθοδο *f()* στη *D1*. Μόλις βρει αυτήν που δηλώνεται στην κλάση θα σταματήσει και δεν θα συνεχίσει την αναζήτηση στη βασική κλάση.

Το πρόβλημα λύνεται αν γράψουμε:

```
   cout << d.B::f() << endl;
```

οπότε και παίρνουμε:

```
B::f()
2
void D1::f()
1  2
```

Ας αλλάξουμε τώρα την *D1* ξαναβάζοντας και την *f()* που είχαμε αρχικώς.

```
class D1: public B
{
public:
   D1( int bp=1, int d1p=0 ): B(bp)  {  md1 = d1p;  }
   virtual int f() const
   {  cout << "D1::f()" << endl; return mb+md1;  }
   virtual void f( int& aV, int& aDV ) const
   {  cout << "void D1::f()" << endl;
      aV = mb;  aDV = md1;  }
private:
   int md1;
}; // D1
```

Τώρα:

- Η "`virtual int f() const`" *υπερισχύει* της *f()* της βασικής αφού έχει ακριβώς την ίδια δήλωση με εκείνη. Τώρα, το "`virtual`" (στην *f()* της *B*) είναι απαραίτητο!
- Η `virtual void f( int& aV, int& aDV ) const` είναι *επιφόρτωση* της προηγούμενης. Το "`virtual`" είναι χρήσιμο μόνο στην περίπτωση που θα γράψουμε συνάρτηση που θα κληρονομήσει τη *D1* (για παράδειγμα τη *DD1*).

Οι

```
   D1 d( 1, 2 );
   cout << d.f() << endl;
   int a1, a2;
```

[15] Borland BC++, v.5.5.

```
   d.f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

δίνουν, χωρίς πρόβλημα:

```
D1::f()
3
void D1::f()
1  2
```

Η συμβουλή λοιπόν είναι: *Δώσε άλλο όνομα στη νέα συνάρτηση*· δεν τελείωσαν τα ονόματα!... Γενικώς:

- ***Απόφευγε να επιφορτώνεις κληρονομούμενες ή υπερισχύουσες (virtual) μεθόδους.***

Σημείωση: ▶

Τώρα, που τα κάναμε όλα σωστά, δοκιμάζουμε το εξής:

```
   B* pD( new D1(1,2) );
   cout << pD->f() << endl;
   int a1, a2;
   pD->f( a1, a2 );
   cout << a1 << "  " << a2 << endl;
```

Ο μεταγλωττιστής μας λέει ότι «η *B* δεν έχει συνάρτηση που να ταιριάζει με την `pD->f(a1, a2)`»!

Αυτό είναι ένα «χαλί» για να περάσουμε στην επόμενη παράγραφο.◀

23.13 Δυναμική Τυποθεώρηση – RTTI

Δες άλλο ένα πρόβλημα παρόμοιο με αυτό που είδαμε στη *Σημείωση* της προηγούμενης παραγράφου. Στην §23.9, όπου είχαμε:

```
   Date* pind[2];
// . . .
   pind[1] = new DateTime( 2003, 3, 3, 3, 33, 33 );
```

μετά την

```
   pind[1]->forward( 10 );
```

ζητήσαμε

```
   cout << pind[1]->getDay() << endl;
```

και –μετά τη διόρθωση με το "`virtual`"– είδαμε ότι η "`pind[1]->getDay()`" μας έδωσε "`3`". Δεν δοκιμάσαμε όμως να δούμε αν αυξήθηκαν τα δευτερόλεπτα σε "`43`". Τώρα που τα ρυθμίσαμε όλα ας σιγουρευτούμε ότι η *forward()* προχώρησε σωστά τα δευτερόλεπτα. Ζητούμε:

```
   cout << pind[1]->getSec() << endl;
```

και ο μεταγλωττιστής μας έχει μια νέα έκπληξη: «`'class Date' has no member named 'getSec'`». Σοκ για τους απρόσεκτους! "`protected`" βάλαμε, "`virtual`" βάλαμε, τι δεν βάλαμε; Όποιος(-α) μελετάει προσεκτικά ξέρει πως αυτά είναι άσχετα με το πρόβλημα:

- Το "`protected`" δίνει πρόσβαση στις μεθόδους της παράγωγης προς μέλη της βασικής.
- Το "`virtual`" δίνει δυνατότητα επιλογής της σωστής *πολυμορφικής* μεθόδου.

Εδώ τι συμβαίνει; Η *getSec()* δεν είναι πολυμορφική (**virtual**) ώστε να δημιουργήσει πρόβλημα επιλογής μεθόδου· είναι απλώς μέθοδος της παράγωγης κλάσης. Θα πρέπει εσύ να βάλεις στο πρόγραμμά σου τις εντολές που θα ερευνήσουν τον τύπο του αντικειμένου που δείχνει το βέλος και να οδηγήσουν στη σωστή εκτέλεση. Πώς διορθώνουμε τα πράγματα; Με **δυναμική τυποθεώρηση** (dynamic casting):

```
   cout << dynamic_cast<DateTime*>(pind[1])->getSec() << endl;
```

Η `dynamic_cast<`*τύπος βέλους*`>(`τιμή-βέλος`)` εξετάζει αν το όρισμα (τιμή-βέλος) δείχνει στην πραγματικότητα μια τιμή του τύπου βέλους που αναφέρεται ως παράμετρος. Αν είναι επιστρέφει βέλος αυτού του τύπου αλλιώς επιστρέφει "`0`" (`NULL`).

Στην περίπτωσή μας αφού βεβαιωθεί ότι το pind[1] δείχνει αντικείμενο κλάσης *DateTime* μας επιστρέφει βέλος τύπου *DateTime** προς το pind[1].

Εδώ εμείς ξέρουμε ότι η μετατροπή του βέλους pind[1] σε βέλος προς αντικείμενο *DateTime* γίνεται σίγουρα (δεν θα πάρουμε "0") και ύστερα από αυτό μπορούμε να ζητήσουμε τη *getSec()*. Λέμε ότι ο τύπος βέλους προς αντικείμενο της βασικής κλάσης είναι **μετατρέψιμος** (convertible) σε τύπο βέλους προς αντικείμενο παράγωγης κλάσης. Αν δεν είχαμε αυτήν τη σιγουριά θα έπρεπε να δουλέψουμε ως εξής:

```
DateTime* temp( dynamic_cast<DateTime*>(pind[1]) );
if ( temp != 0 )
   cout << temp->getSec() << endl;
```

Όπως βλέπεις, η δυναμική τυποθεώρηση βοηθάει να βρούμε απάντηση στο ερώτημα:

- *Κάποιο συγκεκριμένο βέλος δείχνει αντικείμενο κάποιας συγκεκριμένης κλάσης (ή γενικότερα μεταβλητή κάποιου τύπου);*

Αυτό το ερώτημα δεν μπορούμε να το αποφύγουμε όταν χρησιμοποιούμε αυτά που μάθαμε στις προηγούμενες παραγράφους: βέλη της βασικής κλάσης προς αντικείμενα παράγωγης (ή παραγώγων).

Στο παρακάτω κομμάτι προγράμματος βλέπουμε διάφορες περιπτώσεις χρήσης αυτής της τεχνικής:

```
Date* pd = new DateTime( 2004, 4, 4, 4, 44, 44 );
if ( dynamic_cast<Date*>(pd) == 0 )  cout << "OXI" << endl;
                                else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd) == 0 )  cout << "OXI"<<endl;
                                    else  cout << "NAI"<<endl;
Date* pd0 = new Date( 2005, 5, 5 );
if ( dynamic_cast<Date*>(pd0) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
if ( dynamic_cast<DateTime*>(pd0) == 0 )  cout << "OXI"<<endl;
                                     else  cout << "NAI"<<endl;
```

Αποτέλεσμα:

```
NAI
NAI
NAI
OXI
```

Το *pd*, παρ' όλο που έχει δηλωθεί ως βέλος προς αντικείμενο κλάσης *Date*, δείχνει προς αντικείμενο κλάσης *DateTime*. Έτσι,

- Στη δεύτερη περίπτωση, η `dynamic_cast<DateTime*>(pd)` επιστρέφει ως τιμή βέλος τύπου `DateTime*`.
- Στην πρώτη περίπτωση επιστρέφει το ίδιο το *pd*, που δείχνει προς το υποαντικείμενο κλάσης *Date* του αντικειμένου κλάσης `DateTime` που δείχνει το *pd*.
- Πρόσεξε την τελευταία περίπτωση: το *pd*0 δείχνει αντικείμενο κλάσης *Date*. Επομένως δεν μπορεί να μετατραπεί σε βέλος προς αντικείμενο της παράγωγης κλάσης και η `dynamic_cast<DateTime*>(pd0)` επιστρέφει 0.

Παρατήρηση: ▶

Αν έχουμε δηλώσει:

```
DateTime* pdt( new DateTime(2004, 4, 4, 4, 44, 44) );
```

η "`dynamic_cast<Date*>(pdt) != 0`" είναι η διατύπωση σε C++ της ερώτησης: «*δείχνει το pdt αντικείμενο κλάσης Date;*» Φυσικά η απάντηση είναι "`NAI`" και αυτό βγάζει η

```
if ( dynamic_cast<Date*>(pdt) == 0 )  cout << "OXI" << endl;
                                 else  cout << "NAI" << endl;
```

Αν ξαναγυρίσεις στην §23.2 μπορείς να πεις ότι σε ένα αντικείμενο κλάσης *DateTime* το *υποαντικείμενο κλάσης Date** δίνεται από την παράσταση

"`*(dynamic_cast<Date*>(this))`" ◀

Εκτός από τη δυναμική τυποθεώρηση, **εξακρίβωση τύπου κατά την εκτέλεση** (Run Time Type Identification, RTTI) μπορεί να γίνει και με τον τελεστή `typeid`. Ας ξαναθυμηθούμε μερικά πράγματα που μάθαμε στο *Μέρος Α*, §2.8.1: «*μπορείς να δώσεις:*

`typeid(` *παράσταση* `).name()`

και να πάρεις τον τύπο του αποτελέσματος της παράστασης. Στο πρόγραμμά σου θα πρέπει να περιλάβεις (`#include`) *το "*`typeinfo`*".*» Μην αμφιβάλλεις λοιπόν ότι αν δώσεις:

```
cout << typeid(*pd).name() << endl;
cout << typeid(*pd0).name() << endl;
```

θα πάρεις:

```
DateTime
Date
```

Ο τελεστής "`typeid`" επιστρέφει ένα αντικείμενο κλάσης *type_info*, για την οποίαν έχουν επιφορτωθεί και οι τελεστές "`==`" και "`!=`". Έτσι, αν δώσουμε:

```
if ( typeid(*pd) == typeid(Date) )  cout << "NAI" << endl;
                                else  cout << "OXI" << endl;
if ( typeid(*pd) == typeid(DateTime) )  cout << "NAI" << endl;
                                   else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(Date) )  cout << "NAI" << endl;
                                 else  cout << "OXI" << endl;
if ( typeid(*pd0) == typeid(DateTime) )  cout << "NAI" << endl;
                                    else  cout << "OXI" << endl;
```

θα πάρουμε:

```
OXI
NAI
NAI
OXI
```

23.13.1 Μετατροπή Αναφορών

Ας πούμε ότι έχουμε:

```
DateTime dt( 2014, 3, 3, 3, 33, 33 );
cout << "dt = " << dt << endl;
Date& rd( dt );
cout << "rd = " << rd << endl;
```

Αφού όπου μπορούμε να βάλουμε αντικείμενο κλάσης *Date* μπορούμε να βάλουμε και αντικείμενο της παράγωγης κλάσης *DateTime* γράψαμε την εντολή "`Date& rd(dt)`". Αυτές οι εντολές θα δώσουν:

```
dt = 3.3.2014 3:33:33
rd = 3.3.2014
```

«Ε, φυσικά έχουμε τεμαχισμό!» θα πεις. Όχι! Δεν έχουμε τεμαχισμό, αφού δεν έχουμε κάποια αντιγραφή τιμής. Να θυμίσουμε ότι στην §13.4 λέγαμε ότι τα δύο ονόματα –στην περίπτωσή μας *dt* και *rd*– *καθορίζουν την ίδια θέση της μνήμης*. Αυτό που βλέπουμε ως τιμή της *rd* οφείλεται στο ότι γράφεται με τον "`<<`" της *Date*. Πράγματι, αν βάλουμε:

```
DateTime* pRd( reinterpret_cast<DateTime*>(&rd) );
cout << "*pRd = " << *pRd << endl;
```

θα πάρουμε:

```
*pRd = 3.3.2014 3:33:33
```

Θα πεις τώρα «Θα μου τύχει ποτέ να γράψω τέτοια πράγματα στο πρόγραμμά μου;» Όχι βέβαια, αλλά μπορεί να σου τύχει να γράψεις μια συνάρτηση σαν την:

```
void aFunc( Date& aDate )
```

και να θέλεις

- αν κληθεί με όρισμα κλάσης *DateTime* να εκτελεσθούν οι εντολές *EDT* ενώ
- αν κληθει με όρισμα κλάσης *Date* να εκτελεσθούν οι εντολές *ED*.

Ο πάγιος τρόπος χειρισμού του προβλήματος δεν βασίζεται στην ερμηνευτική τυποθεώρηση αλλά στη δυναμική που μπορούμε να την κάνουμε όχι μόνο σε βέλη αλλά και σε τύπους αναφοράς:

```
void aFunc( Date& aDate )
{
// . . .
   try
   {
      DateTime& aDT( dynamic_cast<DateTime&>(aDate) );
   // εντολές EDT για την επεξεργασία του aDT
   }
   catch( bad_cast& )
   {
   // εντολές ED για την επεξεργασία του aDate
   }
// . . .
}
```

Εδώ προσπαθούμε να δούμε –μέσω του τοπικού αντικειμένου *aDT*– την παράμετρο αναφοράς *aDate* ως αντικείμενο κλάσης *DateTime*.

Αν αποτύχει η προσπάθεια ο τελεστής `dynamic_cast` θα ρίξει εξαίρεση κλάσης (*std*::) *bad_cast*. Στο πρόγραμμά σου θα πρέπει έχεις βάλει "`#include <typeinfo>`" όπου υπάρχει ο ορισμός της *bad_cast*.

Εδώ έχουμε μια περίπτωση που η δομή **try/catch** για έγερση και διαχείριση εξαιρέσεων χρησιμοποιείται σαν δομή **if-else**!

23.14 "is_a" ή "has_a"

Στη συνέχεια θα εξετάσουμε ακροθιγώς μερικά σημεία που έχουν σχέση, κατά κύριο λόγο, με αντικειμενοστρεφή σχεδίαση.

Ένας απλουστευτικός τρόπος να δούμε την κληρονομιά είναι και ο εξής: Περιλαμβάνουμε σε κάθε αντικείμενο της παράγωγης ένα αντικείμενο της βασικής. Αυτό όμως μπορούμε να το πετύχουμε πιο απλά. Για παράδειγμα, αντί για:

```
class OfferedCourse :  public Course
{
public:
// . . .
private:
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

θα μπορούσαμε να έχουμε:

```
class OfferedCourse
{
public:
//   . . .
private:
   Course       ocC;
   unsigned int ocNoOfStudents;    // αριθ. Φοιτητών
}; // OfferedCourse
```

Στην περίπτωση αυτή λέμε ότι τα αντικείμενα των δύο κλάσεων συνδέονται με σχέση `has_a` (*OfferedCourse* `has_a` *Course*).

Ποιος τρόπος είναι προτιμότερος; Στο Project 6 ξαναγράφουμε με τον πρώτο τρόπο αυτά που είδαμε στο Project 4. Προσπάθησε να ξαναγράψεις έστω και μέρος του προγράμματος με τον δεύτερο τρόπο και θα καταλάβεις ότι ο πρώτος τρόπος είναι σαφώς προτιμότερος.

Εδώ θα σου δώσουμε ένα πιο απλό παράδειγμα. Έστω ότι έχουμε:

```
class B
```

```
{
public:
   B( int newMb=0 )
      :  mb( newMb ) { };
   virtual ~B() { };
   void f() {  cout << "this is base f" << endl;  };
   virtual void display( ostream& tout )
   {  tout << "base display: mb = " << mb;  };
   virtual void swap( B& other )
   {  std::swap( mb, other.mb );  }
private:
   int mb;
}; // B

class D1 :  public B
{
public:
   D1( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D1() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   virtual void swap( D1& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D1
```

Αν δώσουμε:

```
   D1  x( 3, 5 ), u( 19, 17 );
   x.f();
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
   x.B::display( cout );  cout << endl;
   x.swap( u );
   cout << "after swap" << endl;
   cout << "x: ";  x.display( cout );  cout << endl;
   cout << "u: ";  u.display( cout );  cout << endl;
```

θα πάρουμε:

```
this is base f
x: base display: mb = 3   derived display: md = 5
u: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
x: base display: mb = 19   derived display: md = 17
u: base display: mb = 3   derived display: md = 5
```

Αν θέλουμε να έχουμε την ίδια λειτουργικότητα χωρίς κληρονομιές θα πρέπει να έχουμε:

```
class D2
{
public:
   D2( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   ~D2() { };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  bo.f();  };
   void bdisplay( ostream& tout ) {  bo.display( tout );  };
   void swap( D2& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
private:
```

```
   B   bo;
   int md;
}; // D2
```

Όπως βλέπεις, πρέπει να εξοπλίσουμε τη *D2* με δύο ενδιάμεσες συναρτήσεις –*f()* και *bdisplay()*– που επιτρέπουν την πρόσβαση σε μεθόδους της *B*. Θα πάρουμε τα ίδια αποτελέσματα με την πρώτη περίπτωση δίνοντας:

```
   D2 y( 3, 5 ), t( 19, 17 );
   y.f();
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
   y.bdisplay( cout );  cout << endl;
   y.swap( t );
   cout << "after swap" << endl;
   cout << "y: ";  y.display( cout );  cout << endl;
   cout << "t: ";  t.display( cout );  cout << endl;
```

Παρατήρηση:▶

Αν, παραβιάζοντας τον κανόνα που έχουμε βάλει, μεταφέρουμε τη δήλωση "`B bo;`" στην περιοχή "`public`" τα πράγματα απλουστεύονται κάπως:

```
class D3
{
public:
   D3( int newMb=0, int newMd=0 )
      : md( newMd ) {  bo = B(newMb); };
   void display( ostream& tout )
   {  bo.display( tout );
      tout << "   derived display: md = " << md;  };
   void swap( D3& other )
   {  bo.swap( other.bo );
      std::swap( md, other.md );  }
   B   bo;
private:
   int md;
}; // D3
```

Δίνοντας:

```
   D3 z( 3, 5 ), v( 19, 17 );
   z.bo.f();
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   z.bo.display( cout );  cout << endl;
   z.swap( v );
   cout << "after swap" << endl;
   cout << "z: ";  z.display( cout );  cout << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
```

παίρνουμε τα ίδια αποτελέσματα που πήραμε και πιο πάνω.◀

Ένα καίριο πρόβλημα που δεν φαίνεται στο παράδειγμά μας είναι το εξής: ενώ μπορούμε να δηλώσουμε "`B* p(new D1)`" δεν μπορούμε να δηλώσουμε "`B* p(new D2)`". Στην πρώτη περίπτωση το βέλος προς αντικείμενο της παράγωγης κλάσης "`new D1`" μετατρέπεται σε βέλος προς αντικείμενο της βασικής και στη συνέχεια μπορούμε να βρούμε τον τύπο του αντικειμένου (που δείχνει το βέλος) με δυναμική τυποθεώρηση ή τον τελεστή "`typeid`". Τι θα κάνεις αν προσπαθήσεις να μετατρέψεις την κλάση *CourseCollection* (αλλά και τη *StudentCollection*) του Project 6; Εκεί υπάρχει ένας δυναμικός πίνακας –ο ccArr– που κάθε στοιχείο του είναι τύπου "`Course*`" αλλά μπορεί να δείχνει και αντικείμενο κλάσης *OfferedCourse*. Πώς θα τον αλλάξεις; Μήπως θα μπορούσαμε να χρησιμοποιήσουμε βέλη "`void*`" και ερμηνευτική τυποθεώρηση; Ναι, βέβαια, αλλά δεν είναι και τόσο απλό. Σε ένα τέτοιο βέλος δεν μπορείς να κάνεις απόπαραπομπή αν δεν κάνεις προηγουμένως τυποθεώρηση. Σε ποιον τύπο όμως; Θα πρέπει κάπου να έχεις φυλαγμένη τη σχετική πληροφορία…

Η σχέση "`is_a`" μπορεί να μας λύσει προβλήματα με απλό τρόπο αλλά, από την άλλη μεριά, *οι κλάσεις μιας ιεραρχίας θα πρέπει να έχουν –εκτός από την προγραμματιστική– και*

τη σωστή νοηματική σχέση. Ας πούμε ότι έχεις να γράψεις μια κλάση *Computer* που περιγράφει έναν υπολογιστή ενώ έχεις ήδη μια κλάση *HardDisk* που περιγράφει έναν σκληρό δίσκο. *Μην ξεκινήσεις* με τη λογική *Computer* `is_a` *HardDisk* –ώστε ένα αντικείμενο *Computer* να περιλαμβάνει ένα αντικείμενο *HardDisk*– επειδή συνειδητοποίησες ότι οι διεπαφές (περιοχές "`public`") μοιάζουν πολύ. Η σωστή σχέση είναι *Computer* `has_a` *HardDisk*.

Στη σχέση "`has_a`" θα επανέλθουμε.

Πάντως αν περιορίσεις τις επιλογές σου μεταξύ "`is_a`" και "`has_a`" μπορεί να οδηγηθείς σε σχεδιαστικά λάθη:

Παράδειγμα

Ας πούμε ότι προσθέτουμε στην *OfferedCourse* ένα μέλος επιπλέον, ας το πούμε *ocSemester*, για να κρατούμε και το εξάμηνο που προσφέρθηκε το μάθημα. Αν τώρα, σε ένα πρόγραμμα κάνουμε συγκριτική επεξεργασία της διεξαγωγής των μαθημάτων για τα τελευταία έξη εξάμηνα θα έχουμε εξάδες αντικειμενων *OfferedCourse* που θα έχουν το ίδιο υποαντικείμενο *Course*. Για να το απόφύγουμε μπορούμε

- να βάλουμε στην *OfferedCourse* απλώς ένα βέλος προς αντικείμενο κλάσης *Course* ή
- να βάλουμε στην *OfferedCourse* μόνον τον κωδικό μαθήματος.

Αυτή είναι μια περίπτωση **αντιστοίχισης** (association) **πολλά-προς-ένα** (many-to-one) ή *N* : 1· ούτε "`is_a`" ούτε "`has_a`".

Καλή λοιπόν η κληρονομιά, αλλά δεν είναι κατ' ανάγκη η καλύτερη επιλογή πάντοτε· χρειάζεται λίγη σκέψη πριν τη χρησιμοποιήσουμε.

23.15 "private", "protected" και "public"

Γενικότερα, όταν σχεδιάζεις μια κλάση θα πρέπει να σκεφτείς καλά πριν αποφασίσεις τι θα βάλεις **private**, τι θα βάλεις **protected** και τι θα βάλεις **public**.

- ***Αν ένα μέλος κλάσης είναι*** `private` ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης και β) φίλες κλάσεις ή συναρτήσεις της κλάσης.***
- ***Αν ένα μέλος είναι*** **protected** ***μπορεί να χρησιμοποιηθεί από α) μεθόδους της κλάσης β) φίλες συναρτήσεις της κλάσης γ) μεθόδους των παραγώγων κλάσεων της και δ) φίλες συναρτήσεις παραγώγων κλάσεων.***
- ***Αν ένα μέλος είναι*** **public** ***μπορεί να χρησιμοποιηθεί από οποιαδήποτε συνάρτηση.***

Όταν γράφεις μια παράγωγη κλάση θα πρέπει να αποφασίσεις για τον τρόπο που αυτή κληρονομεί τη βασική:

- ***Αν δηλώσεις "*** `class D: private B` ***" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***).***
- ***Αν δηλώσεις "*** `class D: protected B` ***" τότε τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από α) τις μεθόδους της D και από τις φίλες συναρτήσεις της D β) τις μεθόδους και τις φίλες των παραγώγων κλάσεων της D. Δικαίωμα να μετατροπής ενός*** `D*` ***σε*** `B*` ***(και*** `D&` ***σε*** `B&`***) έχουν μόνον α) οι μεθοδοι της D και οι φίλες συναρτήσεις της D και β) οι μέθοδοι και τις φίλες των παραγώγων κλάσεων της D.***
- ***Αν δηλώσεις "*** `class D: public B` ***" τότε α) τα μέλη της B που είναι*** `public` ***ή*** `protected` ***μπορεί να χρησιμοποιηθούν από οποιαδήποτε συνάρτηση β) οποιαδήποτε συνάρτηση μπορεί να μετατρέψει ένα βέλος*** `D*` ***σε*** `B*` ***και μια αναφορά*** `D&` ***σε*** `B&`***.***

Ας συμπληρώσουμε λοιπόν τον κανόνα που δώσαμε πιο πάνω:

♦ ***Αν έχουμε δηλώσει "`class D: public B`" τότε όπου μπορούμε να βάλουμε, στο πρόγραμμά μας, ένα αντικείμενο κλάσης B μπορούμε να βάλουμε και αντικείμενο κλάσης D.***

Αυτή είναι η ακριβέστερη διατύπωση της *δυνατότητας υποκατάστασης* που είδαμε στην §23.2.

Η κληρονομιά "**`private`**" είναι στην πραγματικότητα ένας τρόπος υλοποίησης της συσχέτισης "**`has_a`**" που είδαμε στην προηγούμενη παράγραφο. Στο πρόγραμμα που γράψαμε εκεί βάζουμε μια ακόμη κλάση:

```
class D4 : private B
{
public:
   D4( int newMb=0, int newMd=0 )
      : B( newMb ),  md( newMd ) { };
   virtual ~D4() { };
   virtual void display( ostream& tout )
   {  B::display( tout );
      tout << "   derived display: md = " << md;  };
   void f() {  B::f();  };
   void bdisplay( ostream& tout ) {  B::display( tout );  };
   void swap( D4& other )
   {  B::swap( other );
      std::swap( md, other.md );  }
private:
   int md;
}; // D4
```

Οι εντολές:

```
   D4 v( 3, 5 ), w( 19, 17 );
   v.f();
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
   v.bdisplay( cout );  cout << endl;
   v.swap( w );
   cout << "after swap" << endl;
   cout << "v: ";  v.display( cout );  cout << endl;
   cout << "w: ";  w.display( cout );  cout << endl;
```

δίνουν:

```
this is base f
v: base display: mb = 3   derived display: md = 5
w: base display: mb = 19   derived display: md = 17
base display: mb = 3
after swap
v: base display: mb = 19   derived display: md = 17
w: base display: mb = 3   derived display: md = 5
```

Να παρατηρήσουμε τα εξής:

- Λέμε παραπάνω: «*τα μέλη της B που είναι "`public`" ή "`protected`" μπορεί να χρησιμοποιηθούν από τις μεθόδους της D.*» Με βάση αυτήν τη δυνατότητα γράφουμε τις μεθόδους *D4*`::`*f()*, *D4*`::`*bdisplay()* και *D4*`::`*swap()*. Δεν γίνεται να χρησιμοποιηθούν κάπως «πιο έξω», ας πούμε στη **`main()`**; Όχι! Αν γράψεις "**`v.f();`**" ο μεταγλωττιστής θα σου πεί: «**`'B::f()' is not accessible in function main()`**». Παρόμοια αντιμετώπιση θα βρεις αν απόπειραθείς να γράψεις "**`v.B::display(cout);`**". Έτσι, χρειάζονται οι ενδιάμεσες συναρτήσεις *D4*`::`*f()*και *D4*`::`*bdisplay()*, όπως ακριβώς και στη *D2*.
- Λέμε παραπάνω: «*μόνον οι μέθοδοι της D και οι φίλες συναρτησεις της D έχουν δικαίωμα να μετατρέψουν ένα* **`D*`** *σε* **`B*`** *(και* **`D&`** *σε* **`B&`***).*» Με βάση αυτήν τη δυνατότητα γράφουμε τη *D4*`::`*swap()* που είναι ολόιδια με τη *D1*`::`*swap()*. Η κλήση "**`B::swap(other);`**" απαιτεί μετατροπή μιας αναφοράς *D4***`&`** (*other*) σε *B***`&`** που περιμένει η *B*`::`*swap()*. Ούτε αυτή η δυνατότητα μπορεί να χρησιμοποιηθεί «πιο έξω». Αν, ας πούμε, έχεις μια συνάρτηση

```
int f1( B& ap )
```

```
{
//...
```

τότε η κλήση:

```
cout << f1( v );
```

δεν είναι δεκτή· «`Cannot initialize 'B &' with 'D4' in function main()`» θα μας πει ο μεταγλωττιστής.

Ακόμη, δεν μπορείς μέσα στο πρόγραμμά σου να γράψεις:

```
B* bp( new D4 );
```

Όπως καταλαβαίνεις η περίπτωση "`class D: private B`" είναι χρήσιμη όταν θέλεις να χρησιμοποιήσεις τη *B* για την *υλοποίηση* των μεθόδων της *D*. Για τον λόγο αυτόν θα δεις να γράφεται "*D* `is_implemented_in_terms_of` *B*" εκτός από "*D* `has_a` *B*".

Τα ίδια ισχύουν σε μεγάλο ποσοστό και για την περίπτωση "`class D: protected B`". Στην περίπτωση αυτήν παράγωγες κλάσεις της *D* βλέπουν αυτά που κληρονόμησε η *D* από τη *B* ως "`protected`".

23.16 Αφηρημένες Κλάσεις

Αν παρατηρήσουμε τα παραδείγματα που δώσαμε μέχρι τώρα βλέπουμε το εξής: *Σε κάθε παραγωγή η βασική κλάση είναι πιο απλή από την παράγωγη*. Αυτός είναι ένας τρόπος να διαχειριζόμαστε την πολυπλοκότητα στα προγράμματα: ασχολούμαστε με τα προβλήματα σταδιακά, οι δυσκολίες έρχονται μια-μια και όχι όλες μαζί. Ας δούμε άλλο ένα παράδειγμα:

Μια εταιρεία παραγωγής/διανομής ηλεκτρικής ενέργειας έχει τρεις κατηγορίες πελατών:

1. *μεγάλες βιομηχανικές μονάδες που χρεώνονται με* 0.03 *ευρώ/ kWh,*
2. *μικρομεσαίες βιοτεχνικές μονάδες που χρεώνονται με* 0.06 *ευρώ/kWh συν μια πάγια χρέωση που είναι διαφορετική για κάθε καταναλωτή και*
3. *οικιακούς καταναλωτές που χρεώνονται με πάγια χρέωση –διαφορετική για κάθε καταναλωτή– συν* 0.06 *ευρώ/kWh για τις πρώτες A kWh και* 0.1 *ευρώ/kWh για την κατανάλωση πέραν της A. Η A καθορίζεται χωριστά για κάθε καταναλωτή.*

Κάθε καταναλωτής, άσχετα από την κατηγορία του, έχει έναν κωδικό αριθμό (θετικός ακέραιος).

Να ορισθεί ιεραρχία κλάσεων για να παρασταθούν τα στοιχεία των πελατών. Κάθε κλάση θα έχει οπωσδήποτε μέθοδο για τον υπολογισμό της χρέωσης του πελάτη.

Ας ξεκινήσουμε από την πρώτη κατηγορία, για την οποίαν γράφουμε μια κλάση, ας την πούμε *Industrial*. Τι χρειάζεται να κρατούμε για έναν πελάτη αυτής της κατηγορίας; Μόνον τον κωδικό αριθμό. Και τι μεθόδους θα χρειαστούμε; Μια μέθοδο για να παίρνουμε τον κωδικό, μια για να τον ορίζουμε και μια για τη χρέωση.

Πάμε τώρα στη δεύτερη κατηγορία (κλάση *SmallMediumEnt*). Εδώ χρειαζόμαστε όλα όσα έχει η πρώτη κλάση και ακόμη ένα μέλος για την πάγια χρέωση. Φυσικά θα χρειαστούμε και δύο μεθόδους για τον χειρισμός αυτού του μέλους.

Στην τρίτη κατηγορία (κλάση *HomeCons*) θα χρειαστούμε όσα έχει η δεύτερη και ακόμη ένα μέλος για την *A* και δύο μεθόδους για τον χειρισμό της.

Επομένως, βασική κλάση θα είναι η (απλούστερη από όλες) *Industrial*. Η επόμενη απλούστερη είναι η κλάση *SmallMediumEnt* που θα παράγεται από την πρώτη. Η κλάση *HomeCons* είναι η πιο πολύπλοκη και θα είναι παράγωγη *SmallMediumEnt*.

Μέθοδος για τη χρέωση θα υπάρχει και στις τρεις κλάσεις (πολυμορφική). Φυσικά δεν είναι ίδια και για τις τρεις κλάσεις.

Φτάσαμε λοιπόν σε μια ιεραρχία:

Industrial ← *SmallMediumEnt* ← *HomeCons*

Αλλά, μπορούμε να πούμε ότι μια "*μικρομεσαία βιοτεχνική μονάδα*" είναι μια "*μεγάλη βιομηχανική μονάδα*" ή ότι ένας "*οικιακός καταναλωτής*" είναι μια "*μικρομεσαία βιοτεχνική μονάδα*"; Όχι φυσικά! Εκείνο που μπορούμε να πούμε είναι ότι: Από προγραμματιστική άποψη

- ο λογαριασμός μιας "*μικρομεσαίας βιοτεχνικής μονάδας*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μεγάλης βιομηχανικής μονάδας*" και
- ο λογαριασμός ενός "*οικιακού καταναλωτή*" έχει τα χαρακτηριστικά και τη συμπεριφορά του λογαριασμού μιας "*μικρομεσαίας βιοτεχνικής μονάδας*".

Δηλαδή, για να βρούμε τη λογική της ιεραρχίας των κλάσεων που βγάλαμε πρέπει *να προχωρήσουμε κατά ένα επίπεδο αφαίρεσης*.

Στο Σχ. 23-6 βλέπεις μια άλλη ιεραρχία κλάσεων για το παράδειγμά μας. Εισάγουμε μια νέα κλάση, την

```
class ElecConsumer
{
public:
   ElecConsumer( int consCodeP );
   unsigned long int getConsCode() const {  return ecConsCode;  }
   void setConsCode( int consCodeP );
   virtual double charge( double consumption ) const = 0;
protected:
   unsigned long int ecConsCode;
}; // ElecConsumer
```

Πρόσεξε όμως ένα ιδιαίτερο χαρακτηριστικό: Η εικονική μέθοδος *charge* δηλώνεται αλλά δεν ορίζεται· αυτό το δείχνουμε με εκείνο το "`= 0`". Λέμε ότι η *charge* είναι μια **γνήσια εικονική μέθοδος** (true virtual method).

Μια κλάση που έχει μια τουλάχιστον γνήσια εικονική μέθοδο λέγεται **αφηρημένη** (abstract) κλάση. Φυσικά, μια τέτοια κλάση, αφού είναι ελλιπής, δεν μπορεί να έχει αντικείμενα. Σε αντιδιαστολή, οι άλλες κλάσεις, που είναι πλήρεις, και έχουν δικά τους αντικείμενα ονομάζονται **συγκεκριμένες** (concrete) κλάσεις.

Χρησιμοποιούμε τις αφηρημένες κλάσεις για να δημιουργούμε "σωστές" ιεραρχίες κλάσεων, όπου φυσικά μια αφηρημένη κλάση (όπως η *ElecConsumer*) είναι βασική κλάση ενώ οι παράγωγές της είναι συγκεκριμένες. Για τον λόγο αυτόν θα τη δεις και ως **αφηρημενη βασική κλάση**. Σε μια ιεραρχία κλάσεων μπορεί να υπάρχουν αφηρημένες κλάσεις σε περισσότερα από ένα στάδια παραγωγής.

Από νοηματική άποψη η νέα ιεραρχία κλάσεων είναι πολύ καλύτερη από την αρχική. Πράγματι, στην αφηρημένη βασική κλάση *ElecConsumer* υπάρχουν τα γενικά χαρακτηριστικά ενός καταναλωτή ηλεκτρικής ενέργειας. Στη συνέχεια, αντί της

$$Industrial \leftarrow SmallMediumEnt \leftarrow HomeCons$$

έχουμε τρεις παραγωγές:

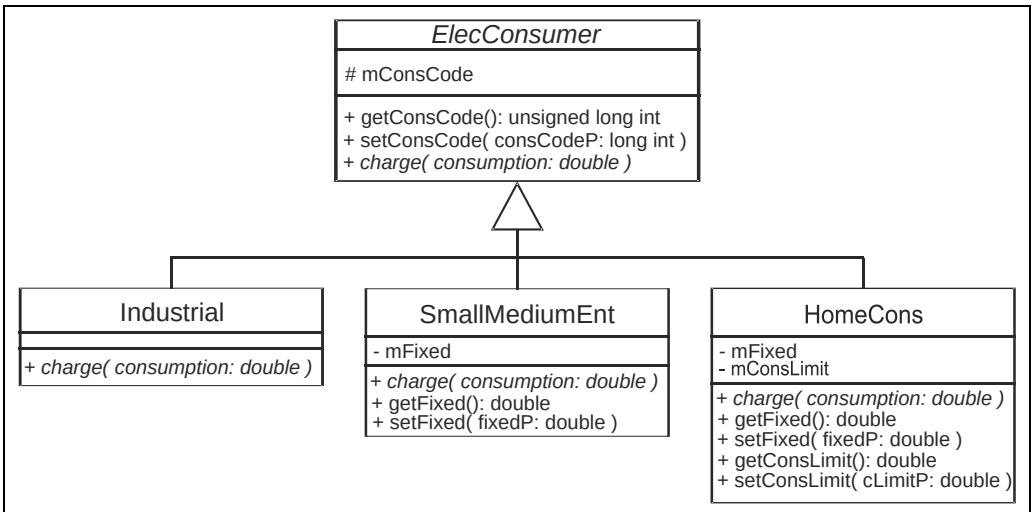
$$ElecConsumer \leftarrow Industrial$$
$$ElecConsumer \leftarrow SmallMediumEnt$$
$$ElecConsumer \leftarrow HomeCons$$

που έχουν νόημα: Ο *βιομηχανικός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*, η *μικρομεσαία επιχείρηση* `is_a` *καταναλωτής ηλεκτρικής ενέργειας* και ο *οικιακός καταναλωτής* `is_a` *καταναλωτής ηλεκτρικής ενέργειας*.

23.17 Η Σειρά Δημιουργίας

Η σειρά δημιουργίας ενός αντικειμένου, που είδαμε εν μέρει στις §3.1.2 και §10.3 μπορεί τώρα να δοθεί ολοκληρωμένη:

- Δημιουργούνται τα υποαντικείμενα των εικονικών βασικών κλάσεων όπως εμφανίζονται στη λίστα κληρονομιάς.



Σχ. 23-6 Οι τρεις αληθινές κλάσεις, *Industrial*, *SmallMediumEnt*, *HomeCons*, κληρονομούν την αφηρημένη βασική κλάση *ElecConsumer*.

- Δημιουργείται το υποαντικείμενο της βασικής κλάσης.
- Δημιουργούνται τα μέλη του αντικειμένου, με τη σειρά που δηλώνονται στην κλάση.
- Εκτελείται το σώμα του δημιουργού.

23.18 Πολλαπλή Κληρονομιά

Έστω ότι έχουμε την κλάση:

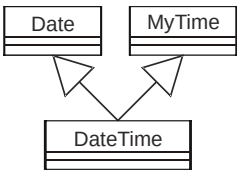
```
class MyTime
{
friend ostream& operator<<( ostream& tout, const MyTime& rhs );
public:
   MyTime( int hp = 0, int minp = 0, int sp = 0 );
   unsigned int getHour() const {  return mtHour;  };
   unsigned int getMin() const {  return mtMin;  };
   unsigned int getSec() const {  return mtSec;  };
   void setHour( int hp );
   void setMin( int minp );
   void setSec( int sp );
   MyTime& operator++();
protected:
   unsigned int mtHour;    // hour (0 .. 23)
   unsigned int mtMin;     // minutes (0 .. 59)
   unsigned int mtSec;     // seconds (0 .. 59)
}; // MyTime
```

Έχουμε επίσης τη γνωστή μας *Date*. Δες έναν άλλον τρόπο να πάρουμε μια κλάση *DateTime*:

```
class DateTime : public Date, public MyTime
{
friend ostream& operator<<( ostream& tout, const DateTime& rhs );
public:
   DateTime( int yp = 1, int mp = 1, int dp = 1,
             int hp = 0, int minp = 0, int sp = 0 )
      : Date( yp, mp, dp ), MyTime( hp, minp, sp ) { };
}; // DateTime

ostream& operator<<( ostream& tout, const DateTime& rhs )
{
   return ( tout << rhs.dDay << '.' << rhs.dMonth << '.' << rhs.dYear << ' '
                 << rhs.mtHour << ':' << rhs.mtMin << ':' << rhs.mtSec );
} // operator<< DateTime
```

Δηλαδή η *DateTime* κληρονομεί δύο κλάσεις: τη *Date* και τη *MyTime*. Αυτό συμβολίζεται διαγραμματικώς όπως βλέπεις στο Σχ. 23-7.



Σχ. 23-7 Η *DateTime* κληρονομεί τη *Date* και τη *MyTime*.

Γενικώς μια παράγωγη κλάση μπορεί να κληρονομεί

- πολλές κλάσεις και
- όχι κατ' ανάγκη με τον ίδιο τρόπο, π.χ.:

```
class X : public A, private B, public C { /* ... */ };
```

Αν δηλώσεις:

```
DateTime dt( 2007, 11, 26, 19, 30, 31 );
```

τότε η *dt* έχει όλες τις ιδιότητες που κληρονόμησε και από τις δύο βασικές κλάσεις! Όλες; Και με τον "`++`" που ορίζεται και στη *Date* και στη *MyTime* τι γίνεται; Εδώ έχουμε μια περίπτωση **ασάφειας** ή **αμφιβολίας** (ambiguity) η οποία λύνεται έτσι:

```
dt.Date::operator++();
cout << dt << endl;
dt.MyTime::operator++();
cout << dt << endl;
```

Αποτέλεσμα:

```
27.11.2007 19:30:31
27.11.2007 19:30:32
```

Η πρώτη, `dt.Date::operator++()`, αυξάνει την ημέρα από 26 σε 27 ενώ η δεύτερη, `dt.MyTime::operator++()`, αυξάνει τα δευτερόλεπτα από 31 σε 32.

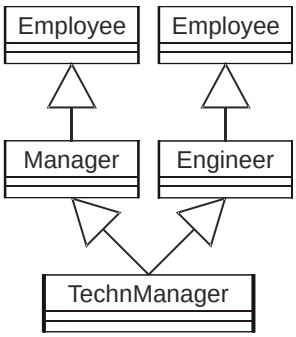
Οι περιπλοκές όμως μπορεί να είναι πιο εντυπωσιακές:

```
class B1 : public L { /* ... */ };
class B2 : public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```

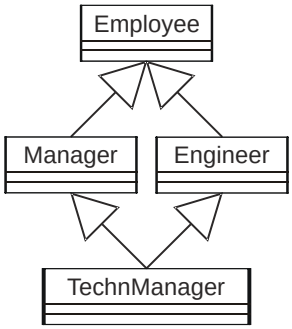
Κάθε αντικείμενο κλάσης *D* έχει δύο υποαντικείμενα κλάσης *L*, το ένα έρχεται από τη *B2* και το άλλο από τη *B3*. Απίθανη περίπτωση; Κάθε άλλο! Βάλε αντί για *L*, *Employee*, αντί για *B1*, *Manager*, αντί για *B2*, *Engineer*, αντί για *D*, *TechnManager* και έχεις το πιο «χειροπιαστό» παράδειγμα στο Σχ.23-8.

Η C++ σου δίνει δυνατότητα να ξεχωρίσεις τα δύο υποαντικείμενα: `B2::L` και `B3::L`. Αλλά το ερώτημα είναι: πόσο χρήσιμο είναι να έχουμε δύο υποαντικείμενα; Στο παράδειγμά μας, του Σχ. 23-8, είναι φανερό ότι για τον Τεχνικό Διευθυντή θέλουμε τα στοιχεία που κρατούμε για κάθε υπάλληλο μια φορά μόνον. Θέλουμε μια εικόνα κληρονομιάς σαν αυτήν του Σχ. 23-9. Αυτό επιτυγχάνεται με την **εικονική** (virtual) κληρονομιά:

```
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ };
```



Σχ. 23-8 Η *TechnManager* κληρονομεί την *Employee* δύο φορές.



Σχ. 23-9 Με κληρονομιά `virtual` η *TechnManager* κληρονομεί την *Employee* μια φορά.

23.19 Διεπαφές

Όπως λέγαμε στην §19.1.3, «*Η οποιαδήποτε επαφή του κάθε προγράμματος με το αντικείμενο γίνεται μέσω αυτών που υπάρχουν στις περιοχές "*`public`*". Αυτά αποτελούν και το τμήμα διεπαφής (interface part) του αντικειμένου.*»

Η Java[16] δίνει την εξής δυνατότητα στον προγραμματιστή: να ορίσει ένα ανεξάρτητο τμήμα διεπαφής και στη συνέχεια να ορίσει τις κλάσεις που το χρησιμοποιούν. Για παράδειγμα, αντιγράφουμε από τα Java Tutorials[17]

```
interface Bicycle
{
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

Αυτή είναι η διεπαφή μιας κλάσης που παριστάνει ποδήλατα. Αν στη συνέχεια θέλουμε μια κλάση για ποδήλατα ACME γράφουμε:

```
class ACMEBicycle implements Bicycle
{
    int cadence = 0;
    int speed = 0;
    int gear = 1;
// . . .
    void speedUp(int increment)
    {   speed = speed + increment;   }
// . . .
}
```

Η C++ δεν μας δίνει αυτήν ακριβώς τη δυνατότητα αλλά μπορούμε να κάνουμε τα ίδια πράγματα χρησιμοποιώντας αφηρημένες κλάσεις. Για το παράδειγμά μας ορίζουμε:

```
class Bicycle
{
public:
   virtual void changeCadence( int newValue ) = 0;
   virtual void changeGear( int newValue ) = 0;
   virtual void speedUp( int increment ) = 0;
   virtual void applyBrakes( int decrement ) = 0;
};

class ACMEBicycle :  public Bicycle
{
public:
   ACMEBicycle()
   {  cadence = 0;  speed = 0;  gear = 1;  }

   virtual void changeCadence( int newValue )
   {  cadence = newValue;  }

   virtual void changeGear( int newValue )
   {  gear = newValue;  }

   virtual void speedUp( int increment )
   {  speed = speed + increment;  }

   virtual void applyBrakes( int decrement )
   {  speed = speed - decrement;  }

   void printStates()
   {  cout << "cadence:" << cadence << " speed:" << speed
```

[16] Παρόμοια δυνατότητα δίνει και η C#.

[17] `http://docs.oracle.com/javase/tutorial/java/concepts/interface.html`

```
          << " gear:" << gear << endl;  }
private:
   int cadence;
   int speed;
   int gear;
}; // ACMEBicycle
```

23.20 Ανακεφαλαίωση

Από μια *βασική κλάση B* μπορούμε να πάρουμε μια *παράγωγη κλάση D* με τη δήλωση:

```
class D :   public B
{ // . . .
```

Η κλάση *D κληρονομεί* τη *B* με την εξής έννοια:

- Όπου μπορούμε να βάλουμε αντικείμενο κλάσης *B* μπορούμε να βάλουμε αντικείμενο κλάσης *D* (*δυνατότητα υποκατάστασης*) αφού:
  - Κάθε αντικείμενο κλάσης *D* έχει όλα τα μέλη και όλες τις συναρτήσεις-μέλη που έχει ένα αντικείμενο κλάσης *B*.
  - Μπορεί να έχει επιπλέον μέλη ή/και συναρτήσεις-μέλη.
- Η κλάση *D κληρονομεί* στατικά μέλη και στατικές συναρτήσεις-μέλη της *B*.
- Δεν κληρονομούνται οι δημιουργοί, ο τελεστής εκχώρησης και ο καταστροφέας της βασικής κλάσης. Μπορείς όμως να τους χρησιμοποιήσεις για να ορίσεις τις αντίστοιχες συναρτήσεις της παράγωγης.

Στην παράγωγη κλάση μπορούμε να ξαναορίσουμε όποιες μεθόδους της βασικής θέλουμε οπότε αυτές *υπερισχύουν* των αντίστοιχων της βασικής. *Στη βασική κλάση οι μέθοδοι που θα ξαναορισθούν πρέπει να είναι δηλωμένες* "`virtual`".

Ορίζοντας τις συναρτήσεις-μέλη της παράγωγης κλάσης μπορεί να χρειαστεί να διαχειριστείς μέλη της βασικής. Ο καλύτερος τρόπος είναι να το κάνεις μέσω των δημιουργών και των μεθόδων "*get*" και "*set*". Αν θεωρείς απαραίτητη την κατ' ευθείαν πρόσβαση στα μέλη δήλωσέ τα (στη βασική) "`protected`" και όχι "`private`".

Λόγω της *δυνατότητας υποκατάστασης*, σε ορισμένες περιπτώσεις χρειάζεται να βρούμε τον τύπο κάποιου αντικειμένου. Η *δυναμική τυποθεώρηση* και ο τελεστής "`typeid`" δίνουν τη λύση σε αυτό το πρόβλημα.

Ερωτήσεις – Ασκήσεις

Α Ομάδα

23-1 Τι θα γράψει το παρακάτω πρόγραμμα:

```
#include <iostream>
class A
{
public:
   A( char c = 'A' )  :  x( c )
   {   cout << "A=" << x; }
   ~A() {  cout << "~A";  }
   A& operator=( const A& a )
   {
      x = a.x;
      cout << "A=" << x;
   }
private:
   char x;
}; // class A
```

```
class C : public A
{
public:
   C( char c = 'C' )  :  z( c )
   {  cout << "C=" << z;  }
   ~C()
   {  cout << "~C";  }
   C& operator=( const C& c )
   {
      A::operator=(c);
      z = c.z;
      cout << "C=" << z;
   }
private:
```

```
class B : public A
{
public:
   B( char c = 'B' )  :  y( c )
   {  cout << "B=" << y;  }
   ~B()
   {  cout << "~B"; }
   B& operator=( const B& b )
   {
      A::operator=( b );
      y = b.y;
      cout << "B=" << y;
   }
private:
   char y;
}; // class B
```

```
   char z;
}; // class C

int main()
{
   cout << "Prints";
   cout << endl << "1:";
   A* b = new B;
   A* c = new C;
   cout << endl << "2:";
   A bb = *b;
   A cc = *c;
   cout << endl << "3:";
   bb = cc;
} // main
```

Β Ομάδα

23-2 Έχουμε τις εξής δύο κλάσεις:

```
class Order
{
public:
. . .
private:
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
}; // Order
```

```
class Auto
{
public:
. . .
private:
   std::string manuf;
   std::string model;
   char        country[3];
   Date        purchDate;
   std::string regNumber;
}; // Auto
```

όπου *Date* η γνωστή κλάση και:

```
struct Good
{
   unsigned int code;
   std::string  unit;
   double       quantity;
   double       price;
}; // Good
```

α) Η πρώτη (**Order**) περιγράφει μια παραγγελία: Ο δυναμικός πίνακας **goods**, με **gCount** στοιχεία, έχει τα αγαθά που παραγγέλνονται. Όπως φαίνεται και από την **struct Good**, για κάθε είδος, έχουμε τον κωδικό του (*code*), τη μονάδα μέτρησης (*unit*) και την ποσότητα (*quantity*) που παραγγέλνεται. Η αξία (*price*) δεν χρησιμοποποιείται. Στην **oDate** έχουμε την ημερομηνία της παραγγελίας ενώ έχουμε και έναν (μοναδικό) αριθμό παραγγελίας (**orderNum**).

β) Η δεύτερη (**Auto**) περιγράφει ένα αυτοκίνητο: τον κατασκευαστή (**manuf**), το μοντέλο (**model**), τη χώρα κατασκευής (**country**, κωδικός χώρας με δύο γράμματα), ημερομηνία αγοράς (**purchDate**), αριθμό (**regNumber**).[18]

Για κάθε μια από τις δύο κλάσεις, άσχετα από την προβλεπόμενη χρήση, παράθεσε (χωρίς υλοποίηση) τους δημιουργούς, τον καταστροφέα και τις μεθόδους / τελεστές που πρέπει να υλοποιήσουμε. Για κάθε ένα από αυτά θα δικαιολογείς γιατί χρειάζεται ή γιατί δεν χρειάζεται η υλοποίηση.

[18] Προφανώς και για τις δύο κλάσεις μπορούμε να σκεφτούμε και άλλα μέλη, απαραίτητα για διάφορες εφαρμογές. Εδώ, θα απαντήσεις τις ερωτήσεις που ακολουθούν με βάση τα μέλη που δίνονται.

Άσχετα από την απάντησή σου στην προηγούμενη ερώτηση γράψε τους δημιουργούς ερήμην και αντιγραφής και για τις δύο κλάσεις. Σχολίασε αυτά που έγραψες και με βάση τις απαντήσεις που έδωσες στην προηγούμενη ερώτηση.

Για την κλάση **Auto** υλοποίησε τον τελεστή εγγραφής ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**).

Για την κλάση **Order** υλοποίησε μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

23-3 Η κλάση:

```
class Invoice
{
public:
. . .
private:
   unsigned int invoiceNum;
   unsigned int orderNum;
   Good*        goods;
   int          gCount;
   std::string  customer;
   Date         oDate;
   Date         iDate;
   double       total;
}; // Order
```

παριστάνει ένα τιμολόγιο. Έχει τον αριθμό της παραγγελίας στην οποίαν αναφέρεται αλλά και δικό του αριθμό τιμολογίου (**invoiceNum**). Έχει την ημερομηνία της παραγγελίας αλλά και την ημερομηνία έκδοσης του τιμολογίου (**iDate**). Τα αγαθά περιγράφονται με τον ίδιο τρόπο, όπως και στην παραγγελία, αλλά τώρα είναι συμπληρωμένη και η αξία (*price*). Τέλος, υπάρχει και η συνολική αξία (*total*).

Όπως έκανες για την **Order**, υλοποίησε και για την **Invoice** μια μέθοδο *print* που θα γράφει την τιμή ενός αντικειμένου σε ένα αρχείο **text** (ή στο **cout**). Για κάθε είδος θα χρησιμοποιείται μια γραμμή.

Η **Invoice** μπορεί να θεωρηθεί ως παράγωγη της **Order**. Γράψε τη δήλωσή της ώστε να φαίνεται αυτή σχέση.

Στην **Order** και στις μεθόδους που έγραψες γι' αυτήν κάνε όλες τις αλλαγές που απαιτούνται ώστε η κληρονόμηση να γίνεται σωστά.

23-4 Έχουμε τις κλάσεις:

```
class A
{
public:
. . .
protected:
   double*      g;
   unsigned int nElmn;
   std::string  c;
}; // A
```

```
class B: public A
{
public:
. . .
private:
   double r;
}; // B
```

(η *nElmn* δείχνει το πλήθος στοιχείων του πίνακα g.)

Συμπλήρωσέ τις με ό,τι χρειάζεται ώστε

α) να μπορώ να γράψω τα εξής:

```
int main()
{
   double q[3] = { 3.0, 1.5, -1.1 };
   A a1;
// a1.nElmn==0 && a1.g==0 && c==""
   A a2( 3, q, "an object" );
// a2.nElmn==3 && a2.g[0]==3.0 && a2.g[1]==1.5 && a2.g[2]==-1.1 &&
// a2.c=="an object"
   A a3( a2 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==1.5 && a3.g[2]==-1.1 &&
```

```
// a3.c=="an object"

   a3.setG( 1, 5.5 );
// a3.nElmn==3 && a3.g[0]==3.0 && a3.g[1]==5.5 && a3.g[2]==-1.1 &&
// a3.c=="an object" &&
// a2.g[1]==1.5 && q[1]==1.5

   B b1( a3, 7.35 );
// b1.nElmn==3 && b1.g[0]==3.0 && b1.g[1]==5.5 && b1.g[2]==-1.1 &&
// b1.c=="an object" && b1.r==7.35
   . . .
```

Στα σχόλια που ακολουθούν κάθε εντολή δίνεται η κατάσταση του θα έχουμε μετά την εκτέλεσή της.

β) Ακόμη, αν έχω δηλώσει:

```
   A*      z[2] = { &a2, &b1 };
   fstream tout( "abc.dta", ios_base::out|ios_base::binary );
```

οι εντολές:

```
   z[0]->write( tout );
   z[1]->write( tout );
```

θα πρέπει να έχουν ως αποτέλεσμα την εγγραφή στον αρχείο `abc.dta`, σε εσωτερική παράσταση (binary), των τιμών των *a*2, *b*1 με την εξής σειρά:

```
3, 3.0, 1.5, -1.1, an object, 3, 3.0, 5.5, -1.1, an object, 7.35
```

23-5 Θέλουμε να γράψουμε μια κλάση:

```
class LongBitMap
{
public:
   typedef unsigned long int  ULong;
   LongBitMap()  {  mBm = new ULong[1];  mBm[0] = 0;
                    mNoOfBits = 0;  mArrSize = 1;  }
   LongBitMap( LongBitMap& other );
   ~LongBitMap() {  delete [] ULong;  }
   long getNoOfBits() const {  return mNoOfBits;  }
   LongBitMap& operator=( LongBitMap& other );
   LongBitMap& operator&=( LongBitMap& other );
   LongBitMap& operator|=( LongBitMap& other );
   LongBitMap& operator^=( LongBitMap& other );
   void lSetBit( int pos );
   void lClearBit( int pos );
   ULong lCount1() const;
   int lBitValue( int pos ) const;
private:
   ULong* mBm;          // δυναμικός ψηφιοπίνακας
   ULong  mNoOfBits;    // πλήθος των bits που χρησιμοποιούμε
   ULong  mArrSize;     // πλήθος στοιχείων (τύπου ULong) του mBm
}; // class LongBitMap
```

για τη διαχείριση μεγάλων ψηφιοπινάκων. Όπως βλέπεις, ο ψηφιοπίνακας παριστάνεται με έναν δυναμικό πίνακα με στοιχεία τύπου `unsigned long int`. Στο mBm[0] υπάρχουν τα bits 0 μέχρι 31, στο mBm[1] τα bits 32 μέχρι 63, στο mBm[2] τα bits 64 μέχρι 95 κ.ο.κ. Το μέλος *mNoOfBits* μας δίνει το πλήθος bits που χρησιμοποιούνται. Το μέλος *mArrSize* μας δίνει το πλήθος των στοιχείων του δυναμικού πίνακα mBm.

Εσύ θα πρέπει να βοηθήσεις στην ολοκλήρωση της υλοποίησης. Να οι προδιαγραφές των μεθόδων που πρέπει να γράψεις:

Δημιουργός Αντιγραφής.

Τελεστής ψηφιακού "**and**" (`operator&=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Τελεστής ψηφιακού "**xor**" (`operator^=`). Αν *mNoOfBits* ≠ *other*.*mNoOfBits* τότε θεωρούμε ότι όλα τα bits που λείπουν από τον «πιο κοντό» ψηφιοχάρτη είναι 0 (μηδέν).

Μέθοδος *lSetBit*. Βάζει 1 στο bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 τότε ο πίνακας επεκτείνεται (όπως μάθαμε στη *renew*) ώστε να περιέχει και θέση *pos*.

Μέθοδος *lBitValue*. Επιστρέφει την τιμή του bit στη θέση *pos*. Αν *pos* > *mNoOfBits*–1 ρίχνει εξαίρεση.

Μπορείς να χρησιμοποιήσεις χωρίς να αντιγράψεις εδώ τα περιγράμματα που συναρτήσεων για ψηφιοχάρτες που υπάρχουν στις σημειώσεις.