κεφάλαιο

24

«Πέφτεις σε Λάθη...» – Εξαιρέσεις

Ο στόχος μας σε αυτό το κεφάλαιο:

Θα ξαναδούμε –συμπληρωμένα– τα εργαλεία που σου προσφέρει η C++ (και η C) για να διαχειριστείς προβλήματα που παρουσιάζονται όταν εκτελείται το πρόγραμμά σου. Θα δεις τις εξαιρέσεις από την καλή και την ανάποδη:

- Θα κατάλάβεις την υπεροχή του μηχανισμού εξαιρέσεων της C++.
- Θα καταλάβεις όμως ότι πρέπει να ασφαλίζεις το πρόγραμμά σου και από τις εξαιρέσεις που το ίδιο ρίχνει.

Προσδοκώμενα αποτελέσματα:

Εξαιρέσεις χρησιμοποιούμε σε μεγάλη έκταση στα μισά από τα προηγούμενα κεφάλαια. Τι καινούριο θα μάθεις εδώ από πρακτική άποψη;

- Κάποια εργαλεία που θα κάνουν τη διαχείριση (των δικών μας) εξαιρέσεων πιο λειτουργική (τουλάχιστον στον καλύτερο εντοπισμό των προβλημλατων).
- Κανόνες για να σχεδιάσεις δικές σου κλάσεις εξαιρέσεων (αν δεν σου αρέσουν αυτές της C++ και οι δικές μας).
- Περισσότερα πράγματα για την ασφάλεια προς τις εξαιρέσεις.

Έννοιες κλειδιά:

- *συνάρτηση assert()*
- *τερματισμός εκτέλεσης*
- *συνάρτηση exit()*
- *προδιαγραφές εξαιρέσεων*
- *συναρτησιακή ομάδα* `try`
- *κλάσεις εξαιρέσεων*
- *ασφάλεια προς τις εξαιρέσεις*

Περιεχόμενα:

24.1 Τι Έχουμε από τη C 919
24.1.1 Η Συνάρτηση *assert()* 919
24.1.2 Συναρτήσεις Τερματισμού Εκτέλεσης 920
24.1.2.1 Σχέση `std::exit()` και `return` 921
24.1.3 Τι Λάθος Έκανα; *errno* 922
24.2 Μήνυμα με Τιμές Συνάρτησης και Παραμέτρων 924
24.3 Συμπληρώματα στην «Ιστορία με Εξαιρέσεις» 927
24.4 Οι Συναρτήσεις Διαχείρισης Εξαιρέσεων 928
24.4.1 Η Συνάρτηση `std::set_terminate()` 928
24.4.2 Η Συνάρτηση `std::terminate()` 930
24.4.3 * Προδιαγραφές Εξαιρέσεων και Σχετικές Συναρτήσεις 933
24.4.4 * Η Συνάρτηση `std::uncaught_exception()` 934
24.5 Συναρτησιακή Ομάδα try 935
24.6 Οι Τύποι Εξαιρέσεων της C++ 936
24.6.1 Η Κλάση *logic_error* και οι Παράγωγές της 938

24.6.1.1 Η Κλάση *domain_error* ... 938
24.6.1.2 Η Κλάση *invalid_argument* ... 939
24.6.1.3 Η Κλάση *length_error* ... 940
24.6.1.4 Η Κλάση *out_of_range* ... 940
24.6.2 Η Κλάση *runtime_error* και οι Παράγωγές της ... 941
24.6.2.1 Η Κλάση *range_error* ... 941
24.6.2.2 Οι Κλάσεις *overflow_error* και *underflow_error* ... 941
24.6.3 Να Χρησιμοποιούμε Αυτές τις Κλάσεις; ... 943
24.7 Πώς να Σχεδιάζεις Δικές σου Κλάσεις Εξαιρέσεων ... 943
24.8 Οι Δικές μας Κλάσεις Εξαιρέσεων ... 944
24.8.1 Δύο Μέθοδοι για τις Κλάσεις Εξαιρέσεων ... 945
24.9 Ασφάλεια ως προς τις Εξαιρέσεις ... 948
24.10 Σύνοψη ... 949

Εισαγωγικές Παρατηρήσεις:

Σε πολλά από τα προγράμματα-παραδείγματα που γράψαμε μέχρι τώρα δίναμε ιδιαίτερη προσοχή στη διασφάλιση κάποιων συνθηκών σε ορισμένα σημεία του προγράμματος. Χαρακτηριστικά παραδείγματα:

- εγκυρότητα των στοιχείων εισόδου που είναι μέρος της εξασφάλισης της προϋπόθεσης του προγράμματος,
- εξασφάλιση της αναλλοίωτης πριν από μια επαναληπτική εντολή,
- εξασφάλιση ότι τα ορίσματα μιας συνάρτησης ανήκουν στο πεδίο ορισμού της (εξασφάλιση προϋπόθεσης),
- εξασφάλιση της αναλλοίωτης μιας κλάσης.

Στην αρχή, μάθαμε να αντιδρούμε «βιαίως» αν βρίσκαμε να μην ισχύει κάποια επιθυμητή συνθήκη. Στην §4.3, στο πρόγραμμα της ελεύθερης πτώσης, είχαμε:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;

   assert( h >= 0 );

// (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
// Υπολόγισε τα tP, vP
```

Αν η συνθήκη-όρισμα της *assert()* πάρει τιμή `false`, η συνάρτηση βγάζει ένα μήνυμα και διακόπτει την εκτέλεση του προγράμματος.

Στο παράδ. 1 της §5.5 γράφαμε κάπως αλλιώς το ίδιο πρόγραμμα:

```
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
   if ( h >= 0 )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX)
   // κάνουμε τους υπολογισμούς
   }
   else
   // false
      cout << " το ύψος δεν μπορεί να είναι αρνητικό" << endl;
```

Αν η τιμή που δίνουμε στην h δεν είναι έγκυρη (αν δηλαδή $h < 0$) τότε βγάζουμε ένα μήνυμα και δεν κάνουμε οτιδήποτε άλλο.

Παρομοίως, στην §7.7 γράφαμε τη

```
   // factorial -- Υπολογίζει το a!
   unsigned long int factorial( int a )
   {
      if ( a < 0 )
      {
         cout << " η factorial κλήθηκε με όρισμα " << a << endl;
         exit( EXIT_FAILURE );
      }
      unsigned long int fv( 1 );
      for ( int k(1);  k <= a;  ++k )  fv *= k;
      return fv;
   } // factorial
```

(ή κάπως έτσι.)

Εδώ, στην περίπτωση που η *factorial()* κληθεί με όρισμα εκτός πεδίου ορισμού (αρνητικό), η συνάρτηση διακόπτει την εκτέλεση του προγράμματος.

Αργότερα μάθαμε να είμαστε πιο ελαστικοί: να δίνουμε τη δυνατότητα στον χρήστη να ξαναδώσει τα (πιθανόν λαθεμένα) στοιχεία εισόδου ή *να ρίχνουμε μια εξαίρεση* αντί να καλέσουμε την *exit()*.

Από τα μέχρι τώρα παραδείγματα θα πρέπει να έχεις αντιληφθεί ότι η αντιμετώπιση των σφαλμάτων στο πρόγραμμα έχει δύο σκέλη:

- Να βρούμε το σφάλμα *εγκαίρως*, πριν μας «κάνει τη ζημιά». Για παράδειγμα, στο πρόγραμμα της ελεύθερης πτώσης (§2.7 και §3.8) αφού η προϋπόθεση λέει ότι θα πρέπει να έχουμε $h \geq 0$ δεν χρειάζεται να φθάσουμε στον υπολογισμό της $\sqrt{2h/g}$.
- Να αντιμετωπίσουμε την κατάσταση. Και πώς γίνεται αυτό;
  - Ένας τρόπος είναι να σταματήσουμε την εκτέλεση του προγράμματος αφού προηγουμένως φυλάξουμε όσα μπορούμε από αυτά που ήδη έκανε. Αυτή η λύση είναι η λιγότερο επιθυμητή.
  - Ένας καλύτερος τρόπος είναι να «καθαρίσουμε» ό,τι έχει «μολυνθεί» από τα λανθασμένα στοιχεία, να φέρουμε το πρόγραμμά μας σε μια αποδεκτή κατάσταση και να συνεχίσουμε από εκεί.

Στο κεφάλαιο αυτό θα συζητήσουμε πιο εκτεταμένα τα εργαλεία που μας δίνει η C++ για να αντιμετωπίζουμε τις «δύσκολες» καταστάσεις. Κατά βάση θα δούμε τις εξαιρέσεις, αλλά πριν από αυτό θα δούμε μερικά εργαλεία που μας δίνει η C, κυρίως αυτά που χρησιμοποιήσαμε ήδη αμέσως ή εμμέσως.

Σημείωση:▶
Πριν προχωρήσουμε θα πρέπει να αναφέρουμε ότι και η C έχει σύστημα εξαιρέσεων με το οποίο όμως δεν θα ασχοληθούμε.◀

24.1 Τι Έχουμε από τη C

Όπως είδαμε η C μας δίνει τη συνάρτηση *assert()* για να ελέγχουμε αν ισχύει κάποια συνθήκη επαλήθευσης σε συγκεκριμένο σημείο του προγράμματός μας.[1] Είδαμε ακόμη τη συνάρτηση *exit()* με την κλήση της οποίας τερματίζεται η εκτέλεση του προγράμματός μας.

24.1.1 Η Συνάρτηση *assert()*

Μπορείς να σκέφτεσαι τη (μακρο)συνάρτηση *assert()* ως:

```
void assert( bool test )
```

Όταν κληθεί, αν το όρισμά της υπολογισθεί σε `true` δεν «αντιδρά». Αν υπολογισθεί `false` τότε δίνει μήνυμα και διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση *abort()*.

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <cassert>`".

[1] Για το νέο πρότυπο της C προτάθηκε (με την τεχνική αναφορά ISO/IEC TR 24731-1-2007) μια οικογένεια εργαλείων για τη διαχείριση παραβίασης περιορισμών (constraints) κατά τη διάρκεια της εκτέλεσης. Σε σχέση με τα εργαλεία αυτά η *assert()* είναι μάλλον απλοϊκή.

24.1.2 Συναρτήσεις Τερματισμού Εκτέλεσης

Η συνάρτηση *exit()*: Την είδαμε για πρώτη φορά στην §7.7 (Παράδ. 1)· προκαλεί κανονικό τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void exit( int status )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση:

- Κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την *atexit()*, που θα δούμε στην συνέχεια. Οι συναρτήσεις εκτελούνται με σειρά αντίστροφη της σειράς καταχώρισης.
- Στη συνέχεια, οι ενταμιευτές των ρευμάτων που έχουν ανοιχθεί για γράψιμο γράφονται στα αντίστοιχα αρχεία και όλα τα ρεύματα κλείνουν.
- Τέλος, τερματίζεται η εκτέλεση του προγράμματος και ο έλεγχος περνάει στο ΛΣ. Αν η τιμή της *status* είναι "`0`" ή *EXIT_SUCCESS* στο ΛΣ περνάει η πληροφορία *επιτυχούς τερματισμού* (successful termination). Αν η τιμή της *status* είναι "`1`" ή *EXIT_FAILURE* περνάει η πληροφορία *ανεπιτυχούς τερματισμού* (unsuccessful termination).[2]

Η συνάρτηση *abort()*: Είπαμε παραπάνω ότι η *assert()* μπορεί να «*διακόπτει την εκτέλεση του προγράμματος καλώντας τη συνάρτηση abort().*» Η *abort()* προκαλεί *μη-κανονικό* τερματισμό της εκτέλεσης του προγράμματος. Μπορείς να τη σκέφτεσαι ως:

```
void abort( )
```

Για να τη χρησιμοποιήσεις θα πρέπει να βάλεις την οδηγία "`#include <cstdlib>`".

Όταν κληθεί η συνάρτηση τερματίζεται η εκτέλεση του προγράμματος και στο ΛΣ περνάει η πληροφορία *ανεπιτυχούς τερματισμού*. Το τι θα γίνει με τα ανοιχτά ρεύματα εξαρτάται από τη συγκεκριμένη υλοποίηση.

Η συνάρτηση *_Exit()*: Η δήλωσή της (στο `cstdlib`) είναι (περίπου):

```
void _Exit( int status );
```

Προκαλεί κανονικό τερματισμό του προγράμματος και περνάει στο ΛΣ πληροφορία όπως η *exit()*. Σε σχέση με τα ρεύματα ισχύουν αυτά που είπαμε για την *abort()*.

Η συνάρτηση *atexit()*: Παραπάνω είπαμε ότι όταν κληθεί η *exit()* «*κατ' αρχάς εκτελούνται όλες οι συναρτήσεις που έχουν καταχωρισθεί με την atexit()*».

Η δήλωσή της *atexit()* (στο `cstdlib`) είναι (περίπου):

```
int atexit( void (*func)() );
```

Δηλαδή, η *atexit()* περιμένει ένα όρισμα που είναι βέλος προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την "`void f()`").

Αν η καταχώριση γίνει επιτυχώς η συνάρτηση επιστρέφει "`0`" ενώ σε περίπτωση αποτυχίας επιστρέφεται μη-μηδενική τιμή. Κάθε υλοποίηση της C++ θα πρέπει να επιτρέπει τουλάχιστον 32 καταχωρίσεις.

Δες ένα παράδειγμα:

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

void bye()
{  cout << "Bye bye!" << endl;  }

void thatsAll()
{  cout << "That is all!!" << endl;  }
```

---

[2] Στο `cstdlib` υπάρχουν οι ορισμοί:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

```
int main()
{
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   if ( atexit(&thatsAll) != 0 )
      cout << "thatsAll registration failed" << endl;

   exit(0);
}
```

Αποτέλεσμα:

```
That is all!!
Bye bye!
```

Πρόσεξε τα εξής:

1. Πρώτα έγινε η καταχώριση της *bye()* και μετά της *thatsAll()*· αλλά κατά τον τερματισμό εκτελείται πρώτα η *thatsAll()* και μετά η *bye()*.
2. Όπως μάθαμε στην §14.3, μπορούμε να μην βάλουμε το "&" και να γράψουμε πιο απλά:

```
   if ( atexit(bye) != 0 ) . . .
   if ( atexit(thatsAll) != 0 ) . . .
```

Προσοχή! Αν στο πρόγραμμά σου δεν βάλεις "`using namespace std`" θα πρέπει να χρησιμοποιείς το πρόθεμα "`std::`" για όλες τις συναρτήσεις που είδαμε παραπάνω.

24.1.2.1 Σχέση `std::exit()` και `return`

Τι διαφορά υπάρχει μεταξύ τερματισμού με *exit()* και τερματισμού με "`return`" στη `main`;

- Αν δώσεις "`return` *n*" θα καταστραφούν –με κλήση των αντίστοιχων καταστροφέων– όλα τα αντικείμενα της `main` που έχουν δημιουργηθεί στη στοίβα και μετά θα εκτελεσθεί η κλήση "`exit( n )`".
- Αν δώσεις "`exit( n )`" δεν θα κληθούν οι καταστροφείς· έτσι, αν περιμένεις κάτι από αυτούς θα πρέπει να φροντίσεις να γίνει με άλλον τρόπο.

Να ένα παράδειγμα:

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

struct B
{
   B( int k ) :  bM( k ) { };
   ~B()
   {  cout << "destruction of a B object: " << bM << endl;  }
private:
   int bM;
}; // B

B b1( 3 );

int main()
{
   B b2( 7 );

   return 0;
}
```

Αυτό το πρόγραμμα θα σου δώσει:

```
destruction of a B object: 7
destruction of a B object: 3
```

Αν αλλάξεις το "`return 0`" σε "`exit( 0 )`" θα πάρεις:

```
destruction of a B object: 3
```

Δηλαδή καλείται ο καταστροφέας να καταστρέψει το *b1* που υλοποιείται σε *στατική* μνήμη αλλά όχι το *b2* που υλοποιείται σε μνήμη στοίβας.[3]

Ωραία όλα αυτά, αλλά αφού και στις δύο περιπτώσεις θα σταματήσει η εκτέλεση του προγράμματος θα «καταστραφούν» τα πάντα! Τι θα μπορούσε να περιμένει από έναν καταστροφέα ένα *καλό* πρόγραμμα; Μια απάντηση μπορεί να είναι η εξής: Να «καθαρίζει» τα αντικείμενα από ευαίσθητες πληροφορίες (ξαναδές την §16.14) πριν επιστραφεί η μνήμη τους στη στοίβα ή στον σωρό!

24.1.3 Τι Λάθος Έκανα; *errno*

Αν έχουμε κάποιο πρόβλημα όταν καλούμε μια συνάρτηση από τις βιβλιοθήκες της C (π.χ.: με λάθος τιμές παραμέτρων) μπορεί να πάρουμε μήνυμα λάθους μέσω της καθολικής μεταβλητής *errno*. Ας ξαναγράψουμε το πρόγραμμα της ελεύθερης πτώσης:

```
#include <iostream>
#include <cmath>
#include <cerrno>
#include <string>
using namespace std;
int main()
{
   const double g( 9.81 ); // m/sec², η επιτάχυνση της βαρύτητας
   double h,   // m, αρχικό ύψος
          tP,  // sec, χρόνος πτώσης
          vP;  // m/sec, ταχύτητα τη στιγμή πρόσκρουσης

// Διάβασε το h
   cout << " Δώσε μου το αρχικό ύψος σε m: ";  cin >> h;
// προσπάθησε να υπολογίσεις το tP
   errno = 0;
   tP = sqrt( (2/g)*h );
   if ( errno != EDOM )
   { // (g == 9.81) && (0 ≤ h ≤ DBL_MAX) && (tP ≈ √(2h/g))
      vP = -g*tP;
   // (tP ≈ √(2h/g)) && (vP ≈ -√(2hg))
   // Τύπωσε τα tP, vP
      cout << " Αρχικό ύψος = " << h << " m" << endl;
      cout << " Χρόνος Πτώσης = " << tP << " sec" << endl;
      cout << " Ταχύτητα τη Στιγμή της Πρόσκρουσης = "
           << vP << " m/sec" << endl;
   }
   else // errno == EDOM
      cout << strerror( errno )
           << ": το ύψος δεν μπορεί να είναι αρνητικό" << endl;
}
```

Να δύο παραδείγματα εκτέλεσης:

```
 Δώσε μου το αρχικό ύψος σε m: 80
 Αρχικό ύψος = 80 m
 Χρόνος Πτώσης = 4.03855 sec
 Ταχύτητα τη Στιγμή της Πρόσκρουσης = -39.6182 m/sec
 Δώσε μου το αρχικό ύψος σε m: -10
Domain error: το ύψος δεν μπορεί να είναι αρνητικό
```

Τι είναι η *errno*; Μπορείς να τη σκέφτεσαι ως μια καθολική μεταβλητή που έχει δηλωθεί «πριν από σένα για σένα» ως[4]

```
int errno;
```

---

[3] Φυσικά, αν δηλώσεις "`static B b2( 7 )`" θα κληθεί ο καταστροφέας και για αυτό.

[4] Το πρότυπο της C λέει ότι μπορεί να είναι κάποιος μακροορισμός που καταλήγει σε μια *τροποποιήσιμη τιμή-l*.

Όταν αρχίζει η εκτέλεση του προγράμματος η τιμή της *errno* είναι "`0`". Ορισμένες συναρτήσεις βάζουν στην *errno* θετική τιμή *αλλά δεν υπάρχει συνάρτηση που να τη μηδενίζει*.

- Ορισμένες συναρτήσεις, μεταξύ αυτών και οι συναρτήσεις της math, αν τροφοδοτηθούν με ορίσματα εκτός πεδίου ορισμού βάζουν στην *errno* τιμή *EDOM* (ορίζεται στο `cerrno`).
- Ορισμένες συναρτήσεις όταν υπολογίσουν εξαιρετικώς μεγάλη τιμή (που δεν παριστάνεται στον τύπο του αποτελέσματος) βάζουν στην *errno* τιμή *ERANGE*.

Τι κάναμε λοιπόν εδώ; Βάλαμε στην *errno* τιμή "`0`" και –χωρίς να ελέγξουμε την τιμή της *h*– δώσαμε "`tP=sqrt((2/g)*h)`". Μετά από αυτό ελέγχουμε την τιμή της *errno* και προχωρούμε μόνο στην περίπτωση που `errno != EDOM`. Ενώ στο αρχικό πρόγραμμα προσέχαμε «*να βρούμε το σφάλμα εγκαίρως, πριν μας "κάνει τη ζημιά"*» εδώ προχωρήσαμε στο επικίνδυνο βήμα και μετά είπαμε «*δεν έπρεπε να το κάνω!*»

Η συνάρτηση *strerror()* παίρνει την τιμή της *errno* και μας δίνει το αντίστοιχο μήνυμα. Στην περίπτωσή μας απεικόνισε το *EDOM* στο "`Domain error`".

Ως δεύτερο παράδειγμα δες το παρακάτω, αφού πρώτα ξαναδείς αυτά που λέγαμε στην §13.8.

```
double d;
char* p;

errno = 0;
d = strtod( "7.1e+318", &p );
if ( (d == HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
errno = 0;
d = strtod( "-7.1e+318", &p );
if ( (d == -HUGE_VAL) && errno == ERANGE )
   cout << "TRUE" << endl;
```

Αποτέλεσμα:

```
TRUE
TRUE
```

Πράγματι, οι τιμές $\pm 7.1\times 10^{+318}$ δεν μπορούν να παρασταθούν στον τύπο `double` και έτσι παίρνουμε ως αποτέλεσμα `±HUGE_VAL` αντιστοίχως ενώ η *errno* παίρνει τιμή *ERANGE*. Το C99 λέει: «*Αν η σωστή τιμή είναι εκτός της περιοχής τιμών που μπορεί να παρασταθούν, επιστρέφεται συν ή πλην HUGE_VAL και η τιμή του ERANGE αποθηκεύεται στην errno.*» Για τον λόγο αυτόν λέγαμε στην §13.8 ότι πρέπει να ελέγχεις τη συνθήκη

```
(d == HUGE_VAL || d == -HUGE_VAL) && errno == ERANGE
```

Γενικώς, οι συναρτήσεις των βιβλιοθηκών της C σε περίπτωση λάθους

- είτε επιστρέφουν τιμή εκτός πεδίου τιμών (όπως η `HUGE_VAL`),
- είτε επιστρέφουν *ακραία* τιμή εντός του πεδίου τιμών (όπως η `INT_MAX`).

ενώ παραλλήλως μπορεί να δίνουν και κάποια θετική τιμή στην *errno*.

Έτσι, η οδηγία που δίνεται για τη χρήση της *errno* είναι η εξής:

- Πριν από την κλήση της συνάρτησης βάλε την εντολή "`errno = 0`".
- Μετά την κλήση της έλεγξε την τιμή που επίστρεψε η συνάρτηση και την τιμή της *errno*.

Πάντως:

- Το πρότυπο δεν δεσμεύει τις διάφορες υλοποιήσεις να επιστρέφουν σε κάθε περίπτωση αποτυχίας συγκεκριμένες τιμές. Για παράδειγμα: δεν λέει τι τιμή θα επιστρέφει η *sqrt()* σε περίπτωση που θα τροφοδοτηθεί με αρνητικό όρισμα.[5]
- Δεν δεσμεύει τις υλοποιήσεις στο να δίνουν τιμή στην *errno*. Για παράδειγμα, για τη *strtod()* το C99 λέει: «*Αν στο αποτέλεσμα έχουμε υποχείλιση (underflow), η συνάρτηση επιστρέφει τιμή που το μέγεθός της δεν υπερβαίνει τον ελάχιστο κανονικοποιημένο*

[5] Οι περισσότερες υλοποιήσεις θα επιστρέψουν `NaN`.

αριθμό του τύπου επιστροφής· το αν η errno θα πάρει τιμή ERANGE καθορίζεται από την υλοποίηση.»[6]

Όπως καταλαβαίνεις, με αυτόν τον τρόπο δεν είναι εύκολο να γράψεις προγράμματα που θα είναι δυνατόν να μεταφερθούν από τη μια υλοποίηση στην άλλη χωρίς αλλαγές.

Τι θα κάνεις λοιπόν;

- ***Προσπάθησε να αποφεύγεις τη χρήση της errno.***
- ***Αν πρέπει να το χρησιμοποιήσεις υποχρεωτικώς συμβουλέψου το C99 και το εγχειρίδιο χρήσης της C που χρησιμοποιείς.***

24.2 Μήνυμα με Τιμές Συνάρτησης και Παραμέτρων

Τα εργαλεία που μας δίνει η C είναι στην πραγματικότητα αυτά που χρειάζονται οι προγραμματιστές που χρησιμοποιούν γλώσσες χωρίς εργαλεία για τη διαχείριση εξαιρέσεων. Οι συναρτήσεις που υπήρχαν στις παλιές βιβλιοθήκες έκοβαν πολύ εύκολα την εκτέλεση του προγράμματος όταν εβρισκαν κάποιο σοβαρό λάθος. Έτσι και οι προγραμματιστές έγραφαν τα προγράμματά τους με τον ίδιο τρόπο· επομένως, η *exit()* και η *abort()* ήταν εργαλεία πολύ χρήσιμα. Αργότερα τα πράγματα άλλαξαν· ο προγραμματιστής δεν ήθελε να του κόβει το πρόγραμμα η *sqrt()* ή κάποια συνάρτηση που έγραφε ο ίδιος ο προγραμματιστής και έτσι η *exit()* και η *abort()* έπαψαν να είναι τόσο χρήσιμες. Αλλά αυτό είχε ως αντίτιμο το να γεμίσει το πρόγραμμα με **if**· και αυτό μπορεί να μην ήταν ενοχλητικό από την άποψη της ταχύτητας εκτέλεσης –οι ταχύτητες των επεξεργαστών είχαν βελτιωθεί– ήταν όμως ενοχλητικό από την άποψη της πολυπλοκότητας του κώδικα καθώς έτσι είχαμε «ανακάτεμα» του αλγόριθμου με τις άμυνες από πιθανά λάθη.

Οι συναρτήσεις έστελναν μηνύματα για τα διάφορα προβλήματα που έβρισκαν με την τιμή της συνάρτησης, με παραμέτρους **out** ή με καθολικές μεταβλητές σαν την *errno*.

Σε πολλές περιπτώσεις, για να μην αυξάνεται το πλήθος των παραμέτρων, η ίδια παράμετρος χρησιμοποιείται για να μεταφέρει όχι μόνον κωδικό λάθους αλλά και άλλες πληροφορίες. Ήδη, εδώ είδαμε:

- Στη συνάρτηση *bisection()*, στην §14.3, επιστρέφουμε μήνυμα λάθους με μια παράμετρο αναφοράς, την *errCode*. Αυτό το είδος λάθους που έχουμε εκεί πιθανότατα απαιτεί ρίψη εξαίρεσης. Ας πούμε όμως ότι κάνεις την αλλαγή που λέγαμε στις παρατηρήσεις και ζητάς να πετύχεις την $|f(x)| < \varepsilon'$ σε *nMax* επαναλήψεις το πολύ (διότι η $|b - a|/2 < \varepsilon$ μπορεί να μη είναι ικανοποιητική). Στην περίπτωση αυτή βάζουμε και άλλη μια παράμετρο, *nMax*, με την οποία βάζουμε έναν μέγιστο αριθμό επαναλήψεων που επιθυμούμε να γίνουν. Πώς θα γνωστοποιήσει η *bisection()* στη συνάρτηση που την καλεί το εξής γεγονός: "*έκανα nMax επαναλήψεις αλλά η $|f(x)| < \varepsilon'$ δεν ισχύει*"; Με εξαίρεση; Όχι βέβαια.
- Στην άσκηση 13-10 σε καλούμε να γράψεις συνάρτηση, την *trinomial()*, που μέσω μιας παρόμοιας παραμέτρου επιστρέφει *το πλήθος των πραγματικών ριζών ή κωδικό λάθους*. Αλλά, ας ξανασκεφτούμε αυτήν την περίπτωση: Το να καλέσουμε τη συνάρτηση για να μας λύσει την εξίσωση "7 = 0" (κωδικός: –1) ή "0 = 0" (κωδικός: *INT_MAX*) σημαίνει πιθανότατα ότι το πρόγραμμά μας έχει κάποιο σοβαρό λάθος. Το να βρούμε ότι το τριώνυμο έχει 0 ή 1 ή 2 πραγματικές ρίζες είναι κάτι το αναμενόμενο. Έτσι, το να εξετάζουμε την τιμή μιας παραμετρου για να καταλάβουμε από την τιμή της αν είναι επιστρεφόμενη τιμή ή κωδικός λάθους μας οδηγεί στο να γράψουμε ένα μπερδεμένο πρόγραμμα. Αν έχεις λύσει την άσκηση, θα έχεις δει ήδη το σχετικό μπέρδεμα και *μέσα* στη συνάρτηση.

[6] Ο μεταγλωττιστής gcc (Dev-C++) βάζει τιμή *ERANGE* στην *errno*.

Θα ήταν λοιπόν καλύτερο να βάλουμε *δύο* παραμέτρους με επιστρεφόμενη τιμή: μια με τον κωδικό σφάλματος και μιαν άλλη για το πλήθος των πραγματικών ριζών. Θα μπορούσαμε ακόμη να μιμηθούμε τη C: να επιστρέφουμε με παράμετρο το πλήθος και να βάζουμε κωδικό λάθους στην *errno* (ή σε κάποια δική μας *myErrNo*.)

Ναι, θα μπορούσαμε να κάνουμε κάτι από αυτά αλλά το μπέρδεμα της διαχείρισης σφαλμάτων με συνήθεις υπολογισμούς παραμένει.

Το πόσο μπερδεμένο μπορεί να είναι ένα πρόγραμμα γραμμένο με αυτόν τον τρόπο φαίνεται όταν έχεις συναρτήσεις που επιστρέφουν μια τιμή. Θα ξαναγράψουμε το πρόγραμμα-παράδειγμα για τους συνδυασμούς που είδαμε στην §14.9.1 με τις εξής απαιτήσεις:

- Κάθε μια από τις συναρτήσεις *comb()*, *factorial()* και *natProduct()* θα πρέπει να αποφεύγει υπολογισμούς με ορίσματα εκτός πεδίου ορισμού, ενώ σε τέτοια περίπτωση θα πρέπει να επιστρέφει κάποιο μήνυμα λάθους. Επειδή οι συναρτήσεις αυτές είναι χρήσιμες γενικώς, θα πρέπει να έχουν μορφή που να τις κάνει χρήσιμες και πέρα από το συγκεκριμένο πρόγραμμα.[7]

Ας πάρουμε τη *natProduct()*. Για αυτήν η προϋπόθεση είναι:

$$0 < m \le n$$

Θα μας δίνει πάντοτε ως αποτέλεσμα έναν θετικό ακέραιο. Για να επιστρέψουμε μήνυμα λάθους

- μπορούμε να βάλουμε μία ακόμη παράμετρο, οπότε θα την κάνουμε (σύμφωνα με τις συνήθειές μας) **void**,

```
// natProduct -- υπολογίζει το m*(m+1)*...*(n-1)*n
void natProduct( int m, int n,
                 unsigned long int& fv, int& err )
{
   if ( m <= 0 || n < m )
      err = 1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
      err = 0;
   }
} // natProduct
```

- μπορούμε όμως να επιστρέφουμε και κάποια αρνητική τιμή:

```
long int natProduct( int m, int n )
{
   long int fv;

   if ( m <= 0 || n < m )
      fv = -1;
   else  // 0 < m <= n
   {
      fv = m;
      for ( int k = m+1; k <= n; ++k )  fv *= k;
   }
   return fv;
} // natProduct
```

Πώς γίνεται τώρα η *factorial()*; Για την πρώτη περίπτωση θα έχουμε:

```
void factorial( int a,
                unsigned long int& fv, int& err )
{
   if ( a == 0 ) {  fv = 1;  err = 0;  }
```

[7] Η τελευταία απαίτηση μας λέει απλούστατα ότι δεν μπορούμε να στηριχθούμε σε έναν έλεγχο στη **main** και να «καθαρίσουμε» για τα πάντα.

```
            else natProduct( 1, a, fv, err );
} // factorial
```

ενώ για τη δεύτερη περίπτωση παραμένει σχεδόν όπως ήταν:

```
long int factorial( int a )
{
   return ( (a == 0) ? 1 : natProduct(1, a) );
} // factorial
```

Το μόνο που άλλαξε είναι ο τύπος του αποτελέσματος –που έγινε `long int`– για να επιτρέπει την επιστροφή της τιμής "`-1`" σε περίπτωση λάθους.

Η εντυπωσιακή αλλαγή γίνεται στην *comb()*:

```
// comb -- Υπολογίζει τους συνδυασμούς των m ανά n
void comb( int m, int n,
           unsigned long int& fv, int& err )
{
   unsigned long int v1, v2;

   if ( n < m-n )
   {
      factorial( n, v1, err );
      if ( err == 0 )
      {
         natProduct( m-n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
   else
   {
      factorial( m-n, v1, err );
      if ( err == 0 )
      {
         natProduct( n+1, m, v2, err );
         if ( err == 0 )  fv = v2/v1;
      }
   }
} // comb
```

Φρίκη! Υπολογισμοί και διαχείριση σφαλμάτων έγιναν «μαλλιά κουβάρια»! Σύγκρινε αυτήν την εκδοχή με την αρχική... Ίδια είναι η κατάσταση και στην περίπτωση που η συνάρτηση επιστρέφει τιμή "`-1`" (άσκηση για σένα!)

Στη **main** θα έχουμε:

```
int main()
{
   int m, n;
   unsigned long int s;
   int  err;

   cout << " Δώσε Δύο Φυσικούς Αριθμούς m, n <= 50, m >= n: ";
   cin >> m >> n;
   comb( m, n, s, err );
   if ( err == 0 )
      cout << " Συνδυασμοί των "
           << m << " ανά " << n << " = " << s << endl;
   else
      cout << " Λάθος δεδομένα" << endl;
} // main
```

Και στις δύο περιπτώσεις, έχουμε χάσει την ευκολία και την κομψότητα της κλήσης μιας συνάρτησης με τύπο.

Να συγκρίνουμε αυτό το πρόγραμμα και την εκτέλεσή του με αυτό που γράψαμε στην §14.9.1:

1. Το πρόγραμμα με την εξαίρεση είναι πολύ πιο απλό και πιο κατανοητό. Η διαφορά έγκειται στη δυνατότητα που έχουμε να καλούμε τις συναρτήσεις (με τύπο) μέσα σε παραστάσεις.
2. Στο πρώτο πρόγραμμα το ξετύλιγμα της στοίβας γίνεται αυτομάτως και πολύ απλά. Εδώ, γίνεται βήμα-προς-βήμα αφού κάθε φορά ελέγξουμε την τιμή της *err*.
3. Το δεύτερο πρόγραμμα έχει γεμίσει με εντολές για τη διαχείριση κωδικών λάθους. Πιο χαρακτηριστική είναι η περίπτωση της *comb()*.

Βλέπουμε λοιπόν δύο σημαντικά πλεονεκτήματα του μηχανισμού διαχείρισης εξαιρέσεων που προσφέρει η C++ (αλλά και άλλες γλώσσες προγραμματισμού):

- Το αυτόματο ξετύλιγμα της στοίβας και η δυνατότητα χρήσης συναρτησεων με τύπο.
- Ο συστηματικός *διαχωρισμός των εντολών διαχείρισης εξαιρετικών συμβάντων* από το υπόλοιπο πρόγραμμα.

Στη συνέχεια θα δούμε και άλλα.

24.3 Συμπληρώματα στην «Ιστορία με Εξαιρέσεις»

Θα δώσουμε τώρα μερικά συμπληρώματα στην «Ιστορία με Εξαιρέσεις» που είδαμε στην §14.9.1. Θα ξεκινήσουμε όμως με έναν ορισμό που αντιγράφουμε από το Java Tutorial:[8] «*Ο όρος εξαίρεση είναι συντομογραφία της φράσης "εξαιρετικό συμβάν."*

- ***Μια εξαίρεση (exception) είναι ένα συμβάν –κατά τη διάρκεια της εκτέλεσης ενός προγράμματος– που διακόπτει την ομαλή ροή των εντολών του προγράμματος.»***

Όταν σε κάποιο σημείο μιας συνάρτησης βρούμε να μην ισχύει κάποια συνθήκη που είναι αναγκαία, **ρίχνουμε** (throw) ή **εγείρουμε** (raise) μιαν εξαίρεση:

"`throw`", *αντικείμενο εξαίρεσης* ;

π.χ.:

```
throw -1;      throw n;      throw x;
```

Συνήθως χρησιμοποιούμε τον όρο *εξαίρεση* αντί για *αντικείμενο εξαίρεσης*.

Τα χαρακτηριστικά μιας εξαίρεσης είναι:

- ο τύπος και
- *η τιμή*

του αντικειμένου της.

Στη συνάρτηση *v()* που είδαμε στην §14.9 έχουμε την εντολή "`throw x;`". Τι θα γίνει αν εκτελεσθεί αυτή η εντολή; Σύμφωνα με αυτά που είδαμε στην §14.9.1, «*Τα σχετικά με την κλήση της» v() «φεύγουν από τη στοίβα και επιστρέφουμε»* στη συνάρτηση που κάλεσε τη *v()*. Και η *x*; Αυτή μας χρειάζεται, αφού είναι το αντικείμενο της εξαίρεσης (και θα πρέπει να γίνει τιμή της παραμετρου της **catch** που θα συλλάβει την εξαίρεση)!

Αυτό που γίνεται είναι το εξής: όταν εκτελεσθεί η **throw** το αντικείμενο της εξαίρεσης αντιγράφεται σε «σίγουρη» θέση όπου ζει μέχρι να τελειώσει η τελευταία **catch** που θα τη συλλάβει (και δεν θα την ξαναρίξει με "`throw;`"). Μετά την εκτέλεση αυτής της **catch** θα κληθεί ο καταστροφέας να καταστρέψει την εξαίρεση.

Αν το αντικείμενο της εξαίρεσης είναι μια σταθερά, όπως το "`-1`" που ρίχνει η *natProduct()*, ή ένα αντικείμενο χωρίς όνομα, όπως "`StudentXptn("Student", StudentXptn::noMemory)`", ο μεταγλωττιστής *μπορεί να κανονίσει* να μη χρειαστεί αντιγραφή (φυλάγοντας καταλλήλως το αντικείμενο της εξαίρεσης εξ αρχής).

Σε κάθε περίπτωση, τα παραπάνω βάζουν την εξής υποχρέωση στον προγραμματιστή που γράφει δικές του κλάσεις εξαιρέσεων:

---

[8] `http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`

- ***Μια κλάση εξαιρέσεων θα πρέπει να έχει σωστό δημιουργό αντιγραφής και σωστό καταστροφέα.***

Όσο η εξαίρεση «ταξιδεύει», κατά το «*ξετύλιγμα της στοίβας*», καλούνται καταστροφείς για να καταστρέψουν τα τοπικά αντικείμενα στις συναρτήσεις που κλείνουν. Όταν εκτελείται κάποιος από αυτούς τους καταστροφείς ή –κάποια συνάρτηση που καλείται από κάποιον καταστροφέα– μπορούμε να ελέγξουμε αν υπάρχει εξαίρεση που δεν έχει συλληφθεί ελέγχοντας την τιμή που επιστρέφει η συνάρτηση (*std*::)*uncaught_exception()*.

Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται αυτομάτως η (*std*::)*terminate()* (που καλεί την (*std*::)*abort()*) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_terminate()*– να αντικαταστήσεις την *terminate()* με δική σου.

Ας πούμε τώρα ότι είχες *προδιαγραφή εξαιρέσεων*:

```
unsigned long int natProduct( int m, int n ) throw( char* )
{
   if ( m <= 0 || n < m )
   {
      throw -1;
   }
// . . .
} // natProduct
```

και εκτελείται η "`throw -1`". Στην περίπτωση αυτήν –και κάθε φορά που ρίχνεται εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση (*std*::) *unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Όπως στην περίπτωση της *terminate()*, η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (*std*::)*set_unexpected()*– να αντικαταστήσεις την (*std*::)*unexpected()* με δική σου.

24.4 Οι Συναρτήσεις Διαχείρισης Εξαιρέσεων

Θα δούμε τώρα τις συναρτήσεις που δίνει η C++[9] για τη διαχείριση εξαιρέσεων. Αλλά να τονίσουμε από την αρχή το εξής: *Αν σχεδιάσεις καλά το προγράμμά σου, αυτές οι συναρτησεις δεν θα σου χρειαστούν*. Αργότερα όμως –όταν οι τροποποιήσεις και οι προσαρμογές στις απαιτήσεις που αλλάζουν– δεν θα καλύπτονται από τις προβλέψεις του αρχικού σχεδίου, αυτές οι συναρτήσεις μπορεί να σε βοηθήσουν να κάνεις διάφορα «μπαλώματα».

24.4.1 Η Συνάρτηση `std::set_terminate()`

Λέγαμε στην προηγούμενη παράγραφο: «*... καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος. Η C++ σου δίνει τη δυνατότητα –με τη συνάρτηση (std::)set_terminate()– να αντικαταστήσεις την terminate() με δική σου.*»

Φαντάσου τώρα ότι η *terminate()*, για την οποίαν τα λέμε στη συνέχεια, είναι κάτι σαν:

```
void terminate()
{
   terminate_handler pDth( &dth );
// . . .
   (*pDth)();
}
```

όπου

```
void dth()
{
// . . .
```

[9] Αναφερόμαστε στη C++03. Η C++11 δίνει και άλλες...

```
   abort();
}
```

και

```
typedef void (*terminate_handler)();
```

Δηλαδή, ο *terminate_handler* είναι τύπος βέλους προς συνάρτηση "`void`" χωρίς παραμέτρους (σαν την παράμετρο της *atexit()*).

Όπως καταλαβαίνεις, η *terminate()* καλεί την *abort() εμμέσως*, μέσω της *dth()*.

Η *set_terminate()* σου επιτρέπει να αλλάξεις τη συνάρτηση που καλεί η *terminate()* και –στο παράδειγμά μας– δείχνει το *pDth*. Για να τη χρησιμοποιήσεις θα θα πρέπει να βάλεις στο πρόγραμμά σου την οδηγία "`#include <exception>`". Στο `exception` μπορείς να δεις την επικεφαλίδα της:

```
terminate_handler set_terminate(terminate_handler f) throw();
```

Ας πούμε ότι γράφουμε τη:

```
void myTerminateHandler()
{
   logFl << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
         << "Φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler
```

και μέσα στο πρόγραμμά μας δίνουμε:

```
   terminate_handler oldTH;
   oldTH = set_terminate( &myTerminateHandler );
```

Τι θα γίνει;

- Ο *pDth* της *terminate()* θα πάρει τιμή "`&myTerminateHandler`", τη διεύθυνση του νέου χειριστή τερματισμού.
- Η *oldTH* θα πάρει ως τιμή αυτήν που είχε πριν η *pDth*, θα δείχνει τον παλιό χειριστή τερματισμού.

Έτσι δουλεύει η *set_terminate()*.[10] Η τιμή που επιστρέφει η συνάρτηση μας είναι συνήθως άχρηστη (εκτός από την περίπτωση που θα θελήσεις να ξαναεγκαταστήσεις αργότερα τον αρχικό χειριστή) και το πιθανότερο είναι να δίνεις απλώς:

```
   set_terminate( &myTerminateHandler );
```

Δες ένα πρόγραμμα που δοκιμάζει τα παραπάνω:

```
#include <fstream>
#include <exception>
#include <cstdlib>

using namespace std;

ofstream logFl( "log.txt" );

void myTerminateHandler()
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main()
{
   set_terminate( &myTerminateHandler );
   throw -1;
}
```

Γιατί βάλαμε τη συνάρτησή μας να γράφει σε αρχείο και όχι στην οθόνη; Για να κάνουμε φανερό ότι αν θέλουμε κάτι τέτοιο θα πρέπει να το κάνουμε με ένα καθολικό αντικείμενο, όπως είναι το ρεύμα *logFl*, αφού η συνάρτησή μας δεν (μπορεί να) έχει παραμέτρους.

---

[10] Τώρα καταλαβαίνεις γιατί λέγαμε «φαντάσου»: εδώ φαίνεται ότι έρχεται η *set_terminate()* να αλλάξει την τιμή μιας τοπικής μεταβλητής (*pDth*) της *terminate()*…

Αν θελήσεις να γράψεις δική σου συνάρτηση για τον χειρισμό του τερματισμού πρόσεξε το εξής:

- ***Η συνάρτηση χειρισμού τερματισμού δεν θα πρέπει να επιστρέφει ως συνήθως με "`return`" αλλά θα πρέπει να τερματίζει την εκτέλεση του προγράμματος, κατά προτίμηση με κλήση της abort().***

Την *abort()* καλεί και η συνάρτηση που βάζει αρχικώς ο μεταγλωττιστής.

24.4.2 Η Συνάρτηση `std::terminate()`

Όπως λέγαμε: «*Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί, καλείται αυτομάτως η (std::)terminate() (που καλεί την (std::)abort()) για να διακόψει την εκτέλεση του προγράμματος.*» Αυτό το βλέπεις και στο παράδειγμα που δώσαμε για τη *set_terminate()*: Στη `main` υπάρχει η "`throw -1`" που θα εκτελεσθεί σίγουρα αλλά δεν υπάρχει **catch** για να τη συλλάβει.

Στη συνέχεια βλέπεις πολλές περιπτώσεις που η *terminate()* καλείται αυτομάτως. Πάντως μπορείς να βάλεις και εντολή κλήσης της στο πρόγραμμά σου αρκεί να βάλεις μια "`#include <exception>`":

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

int main()
{
   set_terminate( &myTerminateHandler );
   terminate();
}
```

Αποτέλεσμα:[11]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

Στο `exception` μπορείς να δεις δήλωση που μοιάζει με:

```
void terminate();
```

Στη συνέχεια παραθέτουμε όλες τις περιπτώσεις που καλείται αυτομάτως η *terminate()* για να διακόψει την εκτέλεση του προγράμματός σου. Πάντοτε η κλήση της γίνεται πριν από το «ξετύλιγμα της στοίβας» εκτός φυσικά από την δεύτερη περίπτωση (δεν βρέθηκε **catch** να συλλάβει την εξαίρεση). Η *std::terminate()* καλείται:[12]

- Όταν ο μηχανισμός διαχείρισης εξαιρέσεων, αφού δημιουργήσει το αντικείμενο της εξαίρεσης και πριν αυτό συλληφθεί από κάποια **catch**, καλέσει κάποια συνάρτηση που ρίχνει εξαίρεση. Ας πούμε ότι έχουμε:

```
struct Xptn
{
```

[11] Borland C++ v.5.5.

[12] Αντιγράφουμε από το πρότυπο…

```
   int c;
   Xptn( int ac=0 )  {  c = ac;  }
   Xptn( const Xptn& rhs ) {  throw -1;  }
}; // Xptn
```

και

```
int main()
{
   Xptn x;
   set_terminate( &myTerminateHandler );
   try {
      throw x;
   }
   catch( Xptn& )
   {  cout << "in catch( Xptn& )" << endl;  }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Αποτέλεσμα:[13]

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Το αντικείμενο που ρίχνει η "`throw x`" είναι έτοιμο. Όταν γίνει προσπάθεια αντιγραφής σε «σίγουρη» θέση –όπως μάθαμε στην §24.3– καλείται ο δημιουργός αντιγραφής. Αυτός ρίχνει άλλη εξαίρεση (`throw -1`) και έτσι καλείται η *terminate()*.

- Όταν έχει ριχτεί μια εξαίρεση, το ξετύλιγμα της στοίβας έχει φτάσει στη **main** και η εξαίρεση δεν έχει συλληφθεί από κάποια **catch**. Ας πούμε ότι έχουμε την *Xptn* και τη *myTerminateHandler()* που είχαμε παραπάνω και ακόμη:

```
void f1()
{  throw Xptn(-1);  }

void f2()
{  f1();  }

int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f2();
   }
   catch( int& )
   {  cout << "in catch( int& )" << endl;  }
}
```

Το πρόγραμμα θα μας δώσει το ίδιο αποτέλεσμα που πήραμε και από το προηγούμενο παράδειγμα. Πώς εξηγείται; Η **main** καλεί την *f2()* και αυτή την *f1()* που ρίχνει εξαίρεση "`Xptn(-1)`". Με το «ξετύλιγμα της στοίβας» επιστρέφουμε στην *f2()* και στη συνέχεια στη **main**. Η εξαίρεση δεν συλλαμβάνεται από την "`catch(int&)`" και αφού δεν υπάρχει "`catch(Xptn&)`" ούτε "`catch(...)`" καλείται η *terminate()*.

- Όταν κατά τη διάρκεια του ξετυλίγματος κληθεί κάποιος καταστροφέας ο οποίος ρίχνει μιαν εξαίρεση. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  }
   ~C()  {  throw 0;  }
};
```

[13] gcc.

```
void f()
{  C lc( 7 );
   throw -1;  }
```

και

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
      f();
   }
   catch( int& )
   {  cout << "int exception" << endl;  }
}
```

Μετά την εκτέλεση της "`throw -1`" καλείται ο καταστροφέας της C για να καταστρέψει το *lc*. Αλλά αυτός ρίχνει εξαίρεση και έτσι καλείται η *terminate()*· το αποτέλεσμα είναι αυτό που είδαμε στα άλλα παραδείγματα.

- Όταν ριχτεί εξαίρεση κατά τη δημιουργία ή την καταστροφή ενός στατικού αντικειμένου. Ας πούμε ότι έχουμε:

```
struct C
{
   int ic;
   C( int ac=0 ) {  ic = ac;  throw -1;  }
};

C gc;
```

Το *gc* είναι ένα καθολικό στατικό αντικείμενο. Με τη εκτέλεση της "`throw -1`" κατά τη δημιουργία του καλείται η *terminate()*.[14] Στην περίπτωση που έχουμε:

```
void f()
{
   static C c( 7 );
}

int main()
{
   set_terminate( &myTerminateHandler );
   f();
}
```

και πάλι, κατά τη δημιουργία του *c*, θα έχουμε κλήση της *terminate()* αλλά τώρα θα δούμε και τους στίχους από το γνωστό άσμα.

- Όταν κατά την εκτέλεση συνάρτησης που έχουμε καταχωρίσει με την *atexit()* ριχτεί εξαίρεση. Αλλάζουμε λίγο τη *bye()* που είδαμε στην §24.1.2:

```
void bye()
{  cout << "Bye bye!" << endl;
   throw -1;  }
```

και δοκιμάζουμε τη

```
int main()
{
   set_terminate( &myTerminateHandler );
   if ( atexit(&bye) != 0 )
      cout << "bye registration failed" << endl;
   exit( 0 );
}
```

Αποτέλεσμα:

```
Bye bye!
```

---

[14] Πριν αρχίσει η εκτέλεση της **main** και επομένως χωρίς να εκτελεσθεί η "`set_terminate( &myTerminateHandler )`".

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

Abnormal program termination
```

- Όταν γίνει απόπειρα εκτέλεσης μιας "`throw;`" (που ξαναρίχνει την εξαίρεση που χειριζόμαστε) ενώ δεν γίνεται διαχείριση κάποιας εξαίρεσης, π.χ.:

```
int main()
{
   set_terminate( &myTerminateHandler );
   throw;
}
```

- Από την *unexpected()*, όπως θα δούμε παρακάτω.

Από τα παραπάνω βγαίνουν και δύο κανόνες προγραμματιστικής πρακτικής· τον πρώτο τον έχουμε ήδη διατυπώσει:

♦ ***Οι καταστροφείς δεν επιτρέπεται να ρίχνουν εξαιρέσεις.***

Θα πεις «δεν μπορώ να φανταστώ περίπτωση που θα έβαζα μια εντολή `throw` σε έναν καταστροφέα.» Σωστό βέβαια, αλλά πρόσεχε αν ο καταστροφέας καλεί άλλες συναρτήσεις. Στην περίπτωση αυτή, ο καταστροφέας θα πρέπει να συλλαμβάνει όλες τις εξαιρέσεις και να σταματάει την μετάδοσή τους.

Τον δεύτερο τον τηρούμε παρ' όλο που δεν τον διατυπώσαμε:

♦ ***Αν γράφεις δικές σου κλάσεις εξαιρέσεων καλό είναι να μη ρίχνονται εξαιρέσεις όχι μόνο από τους καταστροφείς τους αλλά και από τους δημιουργούς τους.***

24.4.3 * Προδιαγραφές Εξαιρέσεων και Σχετικές Συναρτήσεις

Η (*std*::)*unexpected()* καλείται αυτομάτως όταν μια συνάρτηση –που έχει προδιαγραφή εξαιρέσεων– ρίξει εξαίρεση που ο τύπος της δεν υπάρχει στη λίστα τύπων της προδιαγραφής. Η (*std*::)*set_unexpected()* σου δίνει τη δυνατότητα να αλλάζεις την *unexpected()*.

Οι προδιαγραφές εξαιρέσεων έχουν ταλαιπωρήσει αρκετά τους προγραμματιστές που δοκίμασαν να τις χρησιμοποιήσουν. Οι γνώμες ενάντια στο εργαλείο αυτό ήταν πολλές και στο C++11 αποθαρρύνεται η χρήση τους εκτός από την περίπτωση "`throw()`" (η συνάρτηση δεν ρίχνει εξαίρεση). Έτσι λοιπόν και εμείς δίνουμε τη συμβουλή:

♦ ***Μη χρησιμοποιείς προδιαγραφές εξαιρέσεων εκτός από τη "`throw()`" και κατά συνέπεια τις συναρτησεις* unexpected() *και* set_unexpected().**

Πάντως στη συνέχεια θα δούμε εν συντομία τις δύο συναρτήσεις και τη χρήση της κλάσης (*std*::)*bad_exception*.

Ας πούμε ότι έχουμε το πρόγραμμα:

```
#include <iostream>
#include <exception>
#include <cstdlib>

using namespace std;

void myTerminateHandler()
{
   cout << "Φεύγω, κι αφήνω πίσω μου συντρίμμια," << endl
        << "φεύγω, τώρα φεύγω." << endl;
   abort();
} // myTerminateHandler

unsigned long int natProduct( int m, int n ) throw( char* )
{
   throw -1;
} // natProduct
```

```
int main()
{
   set_terminate( &myTerminateHandler );
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
}
```

Αποτέλεσμα:

```
Φεύγω, κι αφήνω πίσω μου συντρίμμια,
φεύγω, τώρα φεύγω.

This application has requested the Runtime to terminate it in an unusual way.
```

Τι έγινε; Η *natProduct()* έριξε εξαίρεση με αντικείμενο "`-1`" (`int`) εκτός προδιαγραφών εξαιρέσεων. Κλήθηκε η *unexpected()* και αυτή με τη σειρά της κάλεσε την *terminate()*.

Τώρα κάνουμε τις εξής αλλαγές:

- Γράφουμε την:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   throw;
}
```

- Αλλάζουμε την προδιαγραφή εξαιρέσεων της *natProduct()*:

```
unsigned long int natProduct( int m, int n ) throw( char*, bad_exception )
{
   throw -1;
} // natProduct
```

- Τέλος, αλλάζουμε και τη **main**:

```
int main()
{
   set_terminate( &myTerminateHandler );
   set_unexpected( &myUnexpected );      // <<<<<<<<<<<<<<
   try {
       cout << natProduct( 3, 1 ) << endl;
   }
   catch( char* )
   {  cout << "catching char*" << endl;  }
   catch( int& )
   {  cout << "catching int" << endl;  }
   catch( bad_exception& )               // <<<<<<<<<<<<<<
   {  cout << "catching bad_exception" << endl;  }
}
```

Αποτέλεσμα:

```
   unexpected; ασ' το σε μένα...
   catching bad_exception
```

Βάζοντας στη λίστα τύπων εξαιρέσεων τον (*std*::)*bad_exception* και γράφοντας τη δική μας *myUnexpected()*, γλυτώσαμε από την *terminate()*!

24.4.4 * Η Συνάρτηση `std::uncaught_exception()`

Η επικεφαλίδα της συνάρτησης, όπως μπορείς να τη δεις στο `exception`, είναι:

```
bool uncaught_exception() throw();
```

Η συνάρτηση αυτή επιστρέφει τιμή "`true`" αν, όταν κληθεί, υπάρχει εξαίρεση που δεν έχει συλληφθεί.

Ας δούμε ένα παράδειγμα χρήσης. Η *myUnexpected()* που γράψαμε παραπάνω έχει το εξής πρόβλημα: Ενώ μπορεί να κληθεί κατ' ευθείαν από το πρόγραμμά μας, σε τέτοια περίπτωση, αν δεν υπάρχει κάποια εξαίρεση «στον αέρα», η "`throw;`" θα προκαλέσει την (αυτοματη) κλήση της *terminate()*. Η σωστή *myUnexpected()* είναι:

```
void myUnexpected()
{
   cout << "unexpected; ασ\' το σε μένα..." << endl;
   if ( uncaught_exception() )
      throw;
}
```

Πού αλλού θα μπορούσε να χρησιμοποιηθεί αυτή η συνάρτηση; Για να την απαντησουμε θα πρέπει να σκεφτούμε ποιες συναρτήσεις ενεργοποιούνται όταν επιστρέφει "`true`" αυτή η συνάρτηση, δηλαδή από τη στιγμή που ρίχτηκε μια εξαίρεση μέχρι να συλληφθεί. Τι γίνεται στο διάστημα αυτό; «Κλείνουν» διάφορες συναρτήσεις και καλούνται καταστροφείς για να καταστρέψουν τα αντικείμενα που ζούσαν μέσα σε αυτές. Επομένως;... Αυτό που κατάλαβες! Μπορείς να τη χρησιμοποιήσεις μόνο μέσα σε καταστροφείς (ή σε συναρτήσεις που αυτοί καλούν). Και πώς θα τη χρησιμοποιήσεις; Θα γράφεις κάτι σαν:

```
   if ( uncaught_exception() )
   {  E1  }
   else
   {  E2  }
```

που σημαίνει: αν έχουμε εξαίρεση «στον αέρα» θα καταστρέψεις το αντικείμενο με τις *E1* αλλιώς θα το καταστρέψεις με τις *E2* (που μπορεί να ρίχνουν και καμιά εξαίρεση).

Ένας προγραμματιστής που σέβεται τον εαυτό του δεν θα γράψει κάτι τέτοιο ακόμη και στην πιο απελπισμένη προσπάθεια να «μπαλώσει» ένα προβληματικό πρόγραμμα.

24.5 Συναρτησιακή Ομάδα try

Στην §23.3 γράψαμε τον δημιουργό της *DateTime* ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp,
                    int hp, int minp, int sp )
   :  Date( yp, mp, dp )
{
// . . .
}; // DateTime::DateTime
```

Αν ο δημιουργός της *Date* ρίξει εξαίρεση εμείς θα πάρουμε το μήνυμα ότι κάποιο πρόβλημα βρήκε ο δημιουργός της *Date*· αυτό όμως είναι ψέμα ή –τουλάχιστον– μισή αλήθεια: Το πρόβλημα βρήκε ο δημιουργός της *Date όταν κλήθηκε από τον δημιουργό της DateTime*.

Τι μπορούμε να κάνουμε για να το διορθώσουμε; Να πιάσουμε την εξαίρεση *DateXptn* από τον δημιουργό της *Date* και να ρίξουμε νέα εξαίρεση *DateTimeXptn*. Αυτό μπορεί να γίνει ως εξής:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  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;
}
catch( DateTimeXptn& x )
```

```
{  throw;  }
catch( DateXptn& x )
{
   throw DateTimeXptn( "DateTime", x.errorCode,
                       x.errVal1, x.errVal2, x.errVal3 );
} // DateTime::DateTime
```

Τι έχουμε εδώ;

- Η ομάδα της **try** ταυτίζεται με το σώμα της συνάρτησης· από εδώ και το όνομα **συναρτησιακή ομάδα try** (functional try-block). Οι **catch** ανήκουν στη συνάρτηση παρ' όλο που φαίνεταο να είναι έξω από αυτήν.
- Η λίστα εκκίνησης ελέγχεται από την **try**, παρ' όλο που βρίσκεται πριν από το "`{`".
- Η πρώτη **catch** πιάνει τις εξαιρέσεις που ρίχνονται από τον ίδιο τον δημιουργό και τις ξαναρίχνει χωρίς άλλη ενέργεια ("`throw;`").
- Η δεύτερη **catch** πιάνει οποιαδήποτε εξαίρεση ριχτεί από τη λίστα εκκίνησης –στην περίπτωσή μας από την κλήση του δημιουργού της βασικής– και την ξαναρίχνει αφού τη μετατρέψει σε *DateTimeXptn*.

Αμφότερες οι ομάδες **catch** τελειώνουν με `throw`. Σύμπτωση; Όχι!

- ***Αν έχεις συναρτησιακή ομάδα try σε δημιουργό όλες οι ομάδες catch που αντιστοιχούν σε αυτήν πρέπει να τελειώνουν με `throw`. Αν δεν υπάρχει τότε ο μεταγλωττιστής θα βάλει ερήμην σου μια "`throw;`".***

Φυσικά, αυτό δεν είναι περίεργο: το ότι ήλθε κάποια εξαίρεση σημαίνει ότι δεν είναι δυνατή η δημιουργία κάποιου μέλους του αντικειμένου (ή του υποαντικειμένου της βασικής κλάσης). Επομένως δεν είναι δυνατόν να δημιουργηθεί το αντικείμενο!

Αυτά που ισχύουν για συναρτησιακή ομάδα **try** *σε δημιουργό ισχύουν και για τον καταστροφέα.* Αφού όμως έχουμε εξηγήσει ότι από τον καταστροφέα δεν επιτρέπεται να ξεφεύγουν εξαιρέσεις, η δυνατότητα να βάλουμε συναρτησιακή ομάδα **try** σε καταστροφέα μας είναι τελείως άχρηστη!

Μπορείς να βάλεις συναρτησιακή ομάδα **try** σε οποιαδήποτε συνάρτηση εκτός από τη **main**. Στην περίπτωση αυτή έχουμε μια διαφορά από αυτά που ισχύουν για δημιουργούς και καταστροφείς: *δεν υπάρχει υποχρέωση για* `throw` *στις αντίστοιχες ομάδες* **catch**. Αλλά και αυτή η δυνατότητα μας είναι άχρηστη αφού

Η

```
void f( . . . )
try {
// . . .
}
catch( . . . )
{
// . . .
}
```

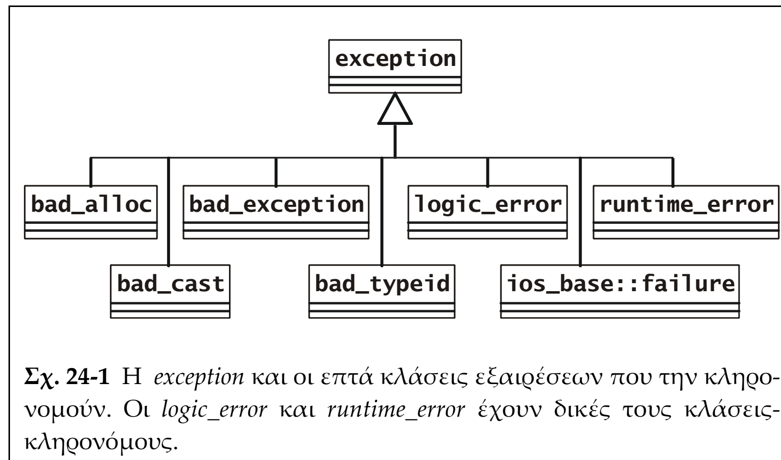
είναι ισοδύναμη με

```
void f( . . . )
{
   try
   {
   // . . .
   }
   catch( . . . )
   {
   // . . .
   }
}
```

24.6 Οι Τύποι Εξαιρέσεων της C++

Η C++ έχει μερικές κλάσεις εξαιρέσεων για χρήση στις βιβλιοθήκες της· οι κλάσεις αυτές είναι διαθέσιμες και στους προγραμματιστές. Σύμφωνα με το πρότυπο της γλώσσας, η κλάση αυτή, που δηλώνεται στο `exception`, πρέπει να είναι:

```
class exception
{
public:
   exception() throw();
   exception( const exception& ) throw();
   exception& operator=( const exception& ) throw();
```



Σχ. 24-1 Η *exception* και οι επτά κλάσεις εξαιρέσεων που την κληρονομούν. Οι *logic_error* και *runtime_error* έχουν δικές τους κλάσεις-κληρονόμους.

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

Όπως βλέπεις, θα πρέπει να έχει:

- ερήμην δημιουργό,
- δημιουργό αντιγραφής,
- τελεστή εκχώρησης,
- καταστροφέα (`virtual`) και
- μια μέθοδο (`virtual`), τη *what()*, που επιστρέφει ένα κείμενο.

Τα πάντα έχουν το χαρακτηριστικό "`throw()`": *δεν ρίχνουν εξαιρέσεις!* Στην §16.6.1 λέγαμε ότι, στις δικές μας κλάσεις εξαιρέσεων, έπρεπε να βάλουμε "`char funcName[100]`" και όχι "`string funcName`" για να έχουμε τη σιγουριά ότι δεν θα ριχτεί εξαίρεση όταν καλείται ο δημιουργός για να δημιουργήσει ή να αντιγράψει ένα αντικείμενο εξαίρεσης. Το πρότυπο λέει: «*Για παράδειγμα, αν το αντικείμενο [εξαίρεσης] είναι κλάσης με δημιουργό αντιγραφής θα κληθεί η* `std::terminate()` *αν η εκτέλεση του δημιουργού αντιγραφής διακοπεί με εξαίρεση όταν εκτελείται μια* `throw`.»

Τι είναι η *what()*; Μια μέθοδος που επιστρέφει ένα κείμενο-μήνυμα. Για παράδειγμα στη Borland C++ v.5.5 το μήνυμα λέει:

```
no named exception thrown
```

Αυτό που είδαμε παραπάνω δεν είναι δήλωση κλάσης· μπορείς να το θεωρήσεις ως διεπαφή.[15] Η κλάση *exception* δεν χρησιμοποιείται στα προγράμματα. Χρησιμοποιούνται οι παράγωγές της που τις βλέπεις στο Σχ. 24-1.

Η πρώτη παράγωγη που ήδη χρησιμοποιούμε είναι η (*std*::)*bad_alloc*. Το πρότυπο λέει:

```
class bad_alloc : public exception
{
public:
   bad_alloc() throw();
   bad_alloc( const bad_alloc& ) throw();
   bad_alloc& operator=( const bad_alloc& ) throw();
   virtual ~bad_alloc() throw();
   virtual const char* what() const throw();
};
```

Όπως βλέπεις, είναι ολόιδια με τη βασική. Βέβαια, εδώ η *what()* βγάζει άλλο μήνυμα (BC++ v.5.5):

```
bad alloc exception thrown
```

Στη C++ που χρησιμοποιείς θα βρεις τη δήλωσή της στο `new`.

Παρόμοια ισχύουν και για τις άλλες παράγωγες κλάσεις, εκτός από την `ios_base::failure`.

[15] Αν ψάξεις στο `exception` μπορεί να βρεις πώς το ολοκληρώνει ως κλάση η C++ που χρησιμοποιείς.

Γνωρίσαμε τη (*std*::)*bad_cast* στην §23.13.1 και στην §Prj06.4. Η δήλωσή της στο `typeinfo`.

Στην §24.4.3 είδαμε την (*std*::) *bad_exception*. Η δήλωσή της στο `exception`.

Η (*std*::)*bad_typeid* ρίχνεται από την *typeid* αν της δώσεις όρισμα (αποπαραπομπή σε) βέλος "`0`". Για να τη χρησιμοποιήσεις πρέπει να βάλεις "`#include <typeinfo>`". Για παράδειγμα, οι:

```
   Student* p( 0 );
   try {
      cout << typeid(*p).name() << endl;
// . . .
   }
   catch ( bad_typeid& x )
   {
      cout << "bad_typeid caught" << endl;
      cout << x.what() << endl;
   }
```

θα δώσουν (g++):

```
bad_typeid caught
St10bad_typeid
```

Εξαιρέσεις κλάσης (*std*::)*ios_base*::*failure* ρίχνονται όταν αποτύχουν μερικές πράξεις διαχείρισης ρευμάτων.[16] Ο ορισμός της κλάσης έχει την εξής διαφορά από τις άλλες κλάσεις που είδαμε: έχει έναν ακόμη δημιουργό

```
   explicit failure( const string& msg );
```

με τον οποίον εισάγουμε (τιμή του *msg*) το κείμενο που μας δίνει η *what()*.

24.6.1 Η Κλάση *logic_error* και οι Παράγωγές της

Η *logic_error* ορίζεται στο `stdexcept`:

```
class logic_error : public exception
{
public:
   explicit logic_error( const string& what_arg );
};
```

Δηλαδή, εκτός από αυτά που έχει η βασική υπάρχει ένας επιπλέον δημιουργός όπως αυτός που είδαμε στην *ios_base*::*failure*.

Όπως λέει το πρότυπο, «*η κλάση logic_error ορίζει τον τύπο των αντικειμένων που ρίχνονται ως εξαιρέσεις για να αναφέρουν σφάλματα υποθετικώς ανιχνεύσιμα πριν από την εκτέλεση του προγράμματος, όπως παραβιάσεις λογικών προϋποθέσεων ή αναλλοίωτων κλάσεων.*»

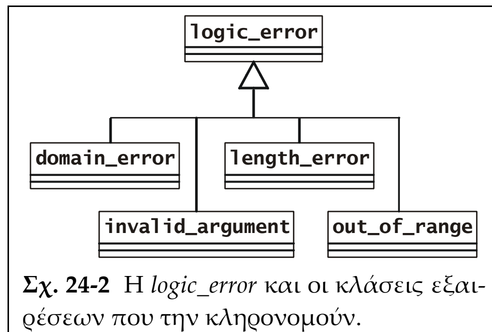
24.6.1.1 Η Κλάση *domain_error*

Η πρώτη παράγωγη κλάση είναι η:

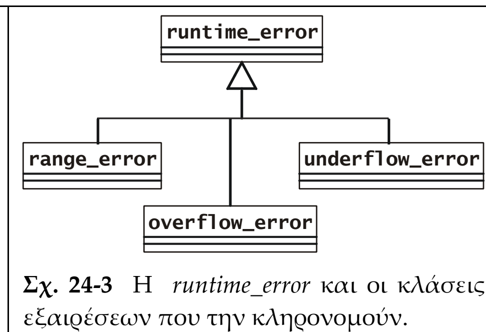
```
class domain_error : public logic_error
{
public:
   explicit domain_error( const string& what_arg );
};
```

Τη ρίχνουμε όταν μια συνάρτηση καλείται με ορίσματα έξω από το **πεδίο ορισμού** (domain of definition) της. Θα μπορούσαμε να γράψουμε τη *factorial()* που είδαμε στις *Εισαγωγικές Παρατηρήσεις* ως εξής:

[16] Στο C++11 η κλάση αυτή είναι παράγωγη της *system_error* που, με τη σειρά της, είναι παράγωγη της *runtime_error*.



Σχ. 24-2 Η *logic_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.



Σχ. 24-3 Η *runtime_error* και οι κλάσεις εξαιρέσεων που την κληρονομούν.

```
unsigned long int factorial( int a )
{
   if ( a < 0 )
   {
      ostringstream ssout;
      ssout << "η factorial κλήθηκε με όρισμα " << a;
      throw domain_error( ssout.str() );
   }
   unsigned long int fv( 1 );
   for ( int k(1);  k <= a;  ++k )  fv *= k;
   return fv;
} // factorial
```

Δημιουργούμε το αντικείμενο της εξαίρεσης με τον δημιουργό της κλάσης συναρμολογώντας το μήνυμα με ένα ρεύμα κλάσης *ostringstream* (§10.12).

Παράδειγμα χρήσης:

```
#include <iostream>
#include <stdexcept>  // για τη domain_error
#include <sstream>    // για το ostringstream

using namespace std;

unsigned long int factorial( int a )
// ΟΠΩΣ ΠΑΡΑΠΑΝΩ

int main ()
{
   try
   {
      cout << factorial( 7 ) << endl;
      cout << factorial( -7 ) << endl;
   }
   catch ( domain_error& x )
   {
      cout << "domain error: " << x.what() << endl;
   }
}
```

Αποτέλεσμα:

```
5040
domain error: η factorial κλήθηκε με όρισμα -7
```

24.6.1.2 Η Κλάση *invalid_argument*

Ορίζεται στο `stdexcept` ως:

```
class invalid_argument : public logic_error
{
public:
   explicit invalid_argument( const string& what_arg );
};
```

Ένα «προβληματικό» όρισμα συνάρτησης συνήθως έχει τιμή εκτός πεδίου ορισμού και το αντιμετωπίζουμε με “`domain_error`”. Αν έχει άλλο πρόβλημα χρησιμοποιούμε την “`invalid_argument`”.

24.6.1.3 Η Κλάση *length_error*

Ορίζεται στο `stdexcept` ως:

```
class length_error : public logic_error
{
public:
   explicit length_error( const string& what_arg );
};
```

Θα μπορούσε να ονομάζεται και “`size_error`”. Τη ρίχνουμε όταν το «μέγεθος» (ό,τι και αν είναι αυτό) κάποιου αντικειμένου υπερβαίνει κάποιο όριο που βάζει η υλοποίηση (ή το πρόγραμμα εφαρμογής). Για παράδειγμα, αν έχουμε δηλώσει:

```
   string s1;
```

οι εντολές:

```
   try {
      s1.reserve( s1.max_size()+1 );
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):[17]

```
std::length_error
invalid string size parameter in function: basic_string::reserve(size_t)
 size: -14 is greater than maximum size: -15
```

24.6.1.4 Η Κλάση *out_of_range*

Ορίζεται στο `stdexcept` ως:

```
class out_of_range : public logic_error
{
public:
   explicit out_of_range( const string& what_arg );
};
```

Θα τη δούμε με ένα παράδειγμα· αν έχουμε:

```
   s1( "test text" );
```

οι εντολές:

```
   try {
      cout << s1.at(17) << endl;
   }
   catch( exception& x )
   {
      cout << typeid( x ).name() << endl
           << x.what() << endl;
   }
```

θα δώσουν (BC++ v.5.5):

```
std::out_of_range
position beyond end of string in function: basic_string::at(size_t)
 index: 17 is greater than max_index: 9
```

Στην πραγματικότητα, τουλάχιστον στην περίπτωση αυτή, πρόκειται για ειδική περίπτωση του “`domain_error`”.

---

[17] Τι είναι τα “`-14`” και “`-15`”; Δεν ξέρεις; Κεφ. 17!

24.6.2 Η Κλάση *runtime_error* και οι Παράγωγές της

Η *runtime_error* ορίζεται στο `stdexcept`:

```
class runtime_error : public exception
{
public:
   explicit runtime_error( const string& what_arg );
};
```

Όπως βλέπεις, κληρονομεί την *exception* με τον ίδιο τρόπο που την κληρονομεί και η *logic_error*.

Σε τι διαφέρει από τη *logic_error*; Η *logic_error* (ή κάποια παράγωγή της) ρίχνεται όταν ξεκινάμε έναν υπολογισμό με τιμή(-ές) κάποιας(-ων) παραμέτρου (-ων) που δεν είναι αποδεκτή(-ές). Η *runtime_error* ρίχνεται κατά τη διάρκεια του υπολογισμού, όταν κάποιο αποτέλεσμα –τελικό ή ενδιάμεσο– έχει κάποιο πρόβλημα.

24.6.2.1 Η Κλάση *range_error*

Η *range_error* είναι παράγωγη της *runtime_error*:

```
class range_error : public runtime_error
{
public:
   explicit range_error( const string& what_arg );
};
```

Ρίχνουμε εξαίρεση αυτού του τύπου όταν έχουμε τιμή συνάρτησης εκτός πεδίου τιμών. Αυτό έχει συχνά να κάνει με τη διαφορά που υπάρχει μεταξύ του πεδίου τιμών της συνάρτησης και του τύπου που παριστάνει αυτό το σύνολο στον υπολογιστή. Για παράδειγμα, αν το πεδίο τιμών μιας συνάρτησης f είναι το $\mathbb{R}$,

f: $D \to \mathbb{R}$

όταν την υλοποιήσουμε με την fc θα το παραστήσουμε με τον `float` ή τον `double` ή τον `long double`:

fc: $Dc \to$ `float` (ή `double` ή `long double`) (C)

Έτσι όμως κάποιες τιμές της συνάρτησης που είναι στο $\mathbb{R}$ μπορεί να μην είναι δυνατόν να παρασταθούν στον τύπο που επιλέξαμε, δηλαδή έχουμε *υπερχείλιση*. Με βάση τη (C) έχουμε πρόβλημα πεδίου τιμών.

Χαρακτηριστικό παράδειγμα είναι η συνάρτηση που *addInt()* που γράψαμε στην §17.3 για να αντιμετωπίσουμε την «ύπουλη» υπερχείλιση στις προσθέσεις ακεραίων. Το σύνολο των ακεραίων $\mathbb{Z}$ είναι κλειστό ως προς την πρόσθεση: το άθροισμα ακεραίων είναι πάντοτε ακέραιος. Αλλά το άθροισμα δύο τιμών τύπου `int` δεν είναι σίγουρο ότι μπορεί να παρασταθεί στον `int`. Εκεί θα μπορούσαμε να ρίχνουμε εξαίρεση τύπου *range_error* αντί για *IntOvrflXptn*.

24.6.2.2 Οι Κλάσεις *overflow_error* και *underflow_error*

Στην *addInt()* θα μπορούσαμε να ρίχνουμε και εξαίρεση κλάσης *overflow_error* που είναι και αυτή παράγωγη της *runtime_error*:[18]

```
class overflow_error : public runtime_error
{
public:
   explicit overflow_error( const string& what_arg );
};
```

Για τις πράξεις στους τύπους κινητής υποδιαστολής υπάρχει και το πρόβλημα της *υποχείλισης* για την οποίαν λέγαμε στην §17.9 «*Συνήθως, ο υπολογιστής σε μια τέτοια περί-*

[18] Πάντως, για ακέραιους τύπους προτιμούμε τον όρο "range error"· ο όρος *υπερχείλιση* αναφέρεται συνήθως σε τύπους κινητής υποδιαστολής.

πτωση θα βάλει το αποτέλεσμα 0 (μηδέν) χωρίς ειδοποίηση για το τι έγινε. Αλλά, αυτό δεν είναι και τόσο τραγικό!»[19] Η κλάση

```
class underflow_error : public runtime_error
{
public:
   explicit underflow_error( const string& what_arg );
};
```

μας δίνεται για τέτοιες περιπτώσεις.

Για να δούμε ένα παράδειγμα χρήσης αυτών των κλάσεων γράφουμε μια *multDbl()* που –ακολουθώντας την *addInt()*– πολλαπλασιάζει δύο τιμές τύπου **double** αλλά έχει πρόβλεψη για υπερχείλιση και υποχείλιση:

```
double multDbl( double x, double y )
{
   double fv;

   if ( fabs(x) >= 1 )
   {
      if ( fabs(y) < 1 )  fv = x * y;
      else if ( fabs(y) <= DBL_MAX / fabs(x) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw overflow_error( ssout.str() );
      }
   }
   else    // |x| < 1
   {
      if ( fabs(y) >= 1 )  fv = x * y;
      else if ( fabs(x) >= DBL_MIN / fabs(y) )  fv = x * y;
      else
      {
         ostringstream ssout;
         ssout << "η multDbl κλήθηκε με ορίσματα "
               << x << ", " << y;
         throw underflow_error( ssout.str() );
      }
   }
   return fv;
} // multDbl
```

Αν δώσεις:

```
  try
  {
     cout << multDbl( DBL_MAX, DBL_MAX ) << endl;
  }
  catch( exception& x )
  {
     cout << typeid(x).name() << endl << x.what() << endl;
  }
```

θα πάρεις (g++, Dev-C++):

```
St14overflow_error
η multDbl κλήθηκε με ορίσματα 1.79769e+308, 1.79769e+308
```

Αν δώσεις:[20]

```
      cout << multDbl( DBL_MIN, DBL_MIN ) << endl;
```

---

[19] Αλλά στην §17.15 δίνουμε δύο παραδείγματα για το πώς να *αποφύγουμε* την υποχείλιση με περισσότερη προσοχή στους υπολογισμούς!

[20] Αν δώσεις απλώς "`cout << DBL_MIN*DBL_MIN << endl`" θα πάρεις "`0`".

θα πάρεις (g++, Dev-C++):

```
St15underflow_error
η multDbl κλήθηκε με ορίσματα 2.22507e-308, 2.22507e-308
```

24.6.3 Να Χρησιμοποιούμε Αυτές τις Κλάσεις;

Όπως είδες, οι κλάσεις εξαιρέσεων της C++ έχουν διαφορετική κατηγοριοποίηση από αυτές που χρησιμοποιούμε εμείς:

- Υπάρχει μια βασική κλάση, η *exception*.
- Από αυτήν παράγονται άλλες κλάσεις για τα διάφορα είδη προβλημάτων.
- Η πληροφορία για το είδος του προβλήματος υπάρχει στο όνομα της κλάσης και οποιαδήποτε άλλα σχετικά στοιχεία διαβιβάζονται μέσω της μεθόδου *what()*.

Αν σε βολεύουν μπορείς να τις χρησιμοποιείς και ακόμη μπορείς να ορίσεις δικές σου παράγωγες κλάσεις για πιο εξειδικευμένα προβλήματα.

24.7 Πώς να Σχεδιάζεις Δικές σου Κλάσεις Εξαιρέσεων

Στις σελίδες των βιβλιοθηκών Boost μπορείς να βρεις οδηγίες για τη διαχείριση λαθών και εξαιρέσεων (Abrahams 2010). Μεταξύ αυτών υπάρχουν και συμβουλές για το πώς να γράφεις δικές σου κλάσεις εξαιρέσεων. Να μια περίληψη:

- **Κάνε τις κλάσεις εξαιρέσεων παράγωγες της `std::exception`**.

  Πρόσεξε τα παραδείγματα που δώσαμε στις προηγούμενες παραγράφους: βάζουμε "`catch(exception& x)`" και με την "`typeid(x).name()`" βρίσκουμε τον τύπο της εξαίρεσης. Έτσι, αν ξέρεις ότι όλες οι κλάσεις εξαιρέσεων του προγράμματός σου είναι παράγωγες της *std::exception* μπορείς να χρησιμοποιείς την "`catch(exception& x)`" αντί για την "`catch(...)`".
- **Χρησιμοποίησε εικονική κληρονομιά** ώστε οι κλάσεις εξαιρέσεων να μπορούν να κληρονομούν δύο ή περισσότερες κλάσεις χωρίς πρόβλημα.

  Ε, τώρα, μη πάρεις και πολύ στα σοβαρά αυτή τη συμβουλή… «*Κλάση εξαιρέσεων που να κληρονομεί δύο ή περισσότερες κλάσεις εξαιρέσεων*»;! Αν τύχει να σου χρειαστεί κάτι τέτοιο μην ξεχάσεις να βάλεις το "`virtual`"!
- **Μη βάζεις στην κλάση εξαιρέσεων μέλη κλάσης `std::string` ή άλλης κλάσης με δημιουργό που μπορεί να ρίξει εξαίρεση**.

  Παρόμοια συμβουλή δώσαμε στην §24.4.2.
- **Μορφοποίησε το μήνυμα της `what()` (μόνο) αν ζητηθεί.** Γενικώς η μορφοποίηση χρειάζεται μνήμη και χρόνο και… μπορεί να φύγει και κάποια εξαίρεση.
- **Μην ασχολείσαι πολύ με το μήνυμα της `what()`**. Είναι καλό βέβαια να δώσεις στον προγραμματιστή τη δυνατότητα να καταλάβει τι συμβαίνει, αλλά είναι μάλλον απίθανο να μπορέσεις να συνθέσεις ένα μήνυμα με τέτοια «προσόντα» όταν ρίχνεις την εξαίρεση.
- **Δώσε πληροφορίες στη διεπαφή (public) της κλάσης εξαιρέσεων για την αιτία του σφάλματος**. Η προσήλωση στο μήνυμα της `what()` πιθανότατα σημαίνει ότι αμελείς την παρουσίαση (χρήσιμων) πληροφοριών.
- **Δώσε στην κλάση εξαιρέσεων «ανοσία» σε διπλή καταστροφή**. Δυστυχώς μερικοί μεταγλωττιστές, σε ορισμένες περιπτώσεις, θα βάλουν εντολές που καταστρέφουν δύο

φορές το αντικείμενο της εξαίρεσης. Αν η κλάση εξαιρέσεων έχει βέλη μην ξεχνάς, στον καταστροφέα, μετά τη “`delete`” να βάλεις και τιμή “`0`” στο κάθε βέλος.[21]

24.8 Οι Δικές μας Κλάσεις Εξαιρέσεων

Στις κλάσεις εξαιρέσεων που χρησιμοποιούμε στα μαθήματά μας έχουμε διαφορετική κατηγοριοποίηση από αυτήν που βλέπουμε στις κλάσεις της C++:

- Μια κλάση εξαιρέσεων για κάθε κλάση. Και ακόμη: αν η κλάση *D* είναι παράγωγη της κλάσης *B* τότε και η κλάση εξαιρέσεων της *D* είναι παράγωγη της κλάσης εξαιρέσεων της *B*.
- Μια κλάση εξαιρέσεων για κάθε βιβλιοθήκη.
- Μια κλάση εξαιρέσεων για κάθε πρόγραμμα.

  Ακολουθούμε τις συμβουλές της προηγούμενης παραγράφου;
- Μέχρι τώρα οι κλάσεις μας δεν είναι παράγωγες της *std::exception*. Από εδώ και πέρα θα είναι! Για παράδειγμα, οι κλάσεις εξαιρέσεων του Project 6 θα είναι:

```
struct CourseXptn :  public exception { /* . . . */ };
struct CourseCollectionXptn :  public exception { /* . . . */ };
struct StudentXptn :  public exception { /* . . . */ };
struct StudentCollectionXptn :  public exception { /* . . . */ };
struct StudentInCourseXptn :  public exception { /* . . . */ };
struct StudentInCourseCollectionXptn :  public exception { /* . . . */ };
struct MyTmpltLibXptn :  public exception { /* . . . */ };
struct ProgXptn :  public exception { /* . . . */ };
```

  Σε όλες αυτές τις κλάσεις θα πρέπει να βάλεις και τον υπερισχύοντα κατάστροφέα, π.χ. για την *CourseXptn*:

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

  Αυτό θα πρέπει να επαναληφθεί και για την *OfferedCourseXptn* που κληρονομεί την *CourseXptn*:

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

  Ακόμη, στη **main()**, αντί για την “`catch(...)`” βάζουμε:

```
catch( exception& x )
{
   cout << "unexpected exception " << typeid(x).name() << endl;
}
```

- Έχουμε επιλέξει να μην βάζουμε στις κλάσεις εξαιρέσεων
  - μέλη-βέλη προς δυναμικές μεταβλητές και
  - μέλη κλάσεων που να χρειάζονται δημιουργούς αντιγραφής.
- Το μορφοποιημένο μήνυμα προς τον χρήστη συντίθεται μετά την σύλληψη του αντικειμένου της εξαίρεσης. Με τη *what()* δεν ασχολούμαστε.
- Όταν ρίχνουμε μια εξαίρεση φροντίζουμε στο αντικείμενό της να υπάρχει
  - το όνομα της συνάρτησης από όπου ρίχτηκε η εξαίρεση και
  - οι «ένοχες» τιμές που την προκάλεσαν.
  - Ακόμη, όταν η εξαίρεση ρίχνεται από μέθοδο ενός αντικειμένου έχουμε και το κλειδί του αντικειμένου.

  Έτσι, δίνουμε αρκετές πληροφορίες για τον εντοπισμό και τη διόρθωση λαθών του προγράμματος. Δίνουμε επίσης τα δεδομένα για διορθωτικές πράξεις κατά τη διάρκεια της εκτέλεσης του προγράμματος, αν κάτι τέτοιο είναι δυνατό.

---

[21] Αυτή η συμβουλή αμφισβητείται διότι δεν φαίνεται να υπάρχουν –τώρα πια– μεταγλωττιστές που κάνουν «διπλή καταστροφή» του αντικειμένου της εξαίρεσης.

Αυτά σε σχέση με τις συμβουλές της προηγούμενης παραγράφου. Θα πρέπει όμως να βελτιώσουμε περισσότερο τις κλάσεις εξαιρέσεων. Ήδη στην §19.4 είχαμε επισημάνει:

- «*Η εξαίρεση θα μας φέρει τη μέθοδο που εμφανίστηκε το πρόβλημα, το είδος του προβλήματος και τις τιμές που το προκάλεσαν ... αλλά δεν θα μας πει σε ποιο αντικείμενο έγιναν όλα αυτά. ... Μια πρώτη σκέψη είναι να αντιγράφουμε στην εξαίρεση και το κλειδί του αντικειμένου που έχει το πρόβλημα. Αλλά πολύ συχνά αυτό δεν είναι αρκετό. ... Αν εξοπλίσουμε κάθε αντικείμενο με ένα επιπλέον μέλος, μια ταυτότητα αντικειμένου που θα την αντιγράφουμε και στο αντικείμενο της εξαίρεσης λύνουμε το πρόβλημά μας.*»
- «*Το δεύτερο πρόβλημα μπορεί να το έχεις αντιμετωπίσει ήδη αν έγραψες κάπως μεγάλα προγράμματα: "Ναι, η εξαίρεση ρίχτηκε από τη συνάρτηση myFunc, αλλά η myFunc καλείται στο πρόγραμμά μου σε 37 διαφορετικές «διαδρομές» εκτέλεσης!"*»

Ήδη, στο Project 6, είδαμε αντικείμενα εξαιρέσεων με το κλειδί του αντικειμενου· πάντως, όπως υποσχεθήκαμε και στην §19.4 *δεν θα δώσουμε παράδειγμα με ταυτότητα αντικειμένου*.

24.8.1 Δύο Μέθοδοι για τις Κλάσεις Εξαιρέσεων

Θα ασχοληθούμε όμως με το δεύτερο πρόβλημα ξεκινώντας από ένα παράδειγμα της §21.6.1. Εκεί, γράφοντας την επιφόρτωση του "`operator=`"της *BString* με βάση τον δημιουργό αντιγραφής, είχαμε γράψει:

```
      try {  BString tmp( rhs );
             swap( tmp );  }
      catch( BStringXptn& x )
      {  strcpy( x.funcName, "operator=" );
         throw;  }
```

Δηλαδή: πιάναμε την όποια εξαίρεση *BStringXptn*, διορθώναμε το *funcName* και την ξαναρίχναμε. Έτσι, φαίνεται ότι την εξαίρεση τη ρίχνει ο "`operator=`".

Το σωστό όμως θα ήταν να πούμε ότι *την εξαίρεση τη ρίχνει ο δημιουργός όταν καλείται από τον* "`operator=`". Θα μπορούσαμε να γράψουμε:

```
      catch( BStringXptn& x )
      {  strcat( x.funcName, "|operator=" );
         throw;  }
```

Έτσι, η *x.funcName* έχει τιμή `"BString|operator="`. Και αν εξοπλίσουμε τη *BStringXptn* με μια:

```
   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                             else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
```

από τις

```
// . . .
         case BStringXptn::noMemory:
            cout << "out of memory "; x.displayFuncName( cout );
            break;
// . . .
```

θα μπορούσαμε να πάρουμε:

```
out of memory thrown in BString
 called by operator=
```

Καλό θα είναι να εξοπλίσουμε τη *BStringXptn* και με μια

```
   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
```

```
      strcat( funcName, fn );  } // appendFuncName
```

και έτσι θα γράφουμε:

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

Η *BStringXptn* τώρα έχει γίνει:

```
struct BStringXptn :  public exception
{
   static const int fnLength = 100;
   enum { noMemory, outOfRange };
   char funcName[ fnLength ];
   int  errorCode;
   int  errorValue;
   BStringXptn( const char* fn, int ec, int ev = 0 )
   {  strncpy( funcName, fn, fnLength-1 );
                                  funcName[fnLength-1] = '\0';
      errorCode = ec;  errorValue = ev;  }
   virtual ~BStringXptn() { };

   void appendFuncName( const char* fn )
   {  strcat( funcName, "|" );
      strcat( funcName, fn );  } // appendFuncName

   void displayFuncName( ostream& tout )
   {
      tout << "thrown in ";
      for ( int k(0); funcName[k] != '\0'; ++k )
         if ( funcName[k] != '|' )  tout << funcName[k];
                              else  tout << endl << " called by ";
      tout << endl;
   } // displayFuncName
}; // BStringXptn
```

Εκτός από τις δύο μεθόδους πρόσεξε και κάτι άλλο: Αντικαταστήσαμε τη «μαγική σταθερά» “`100`” με “`fnLength`” που ορίζεται ως:

```
   static const int fnLength = 100;
```

Αν σκεφτείς ότι μπορεί να χρειαστεί να αποθηκεύεις στη *funcName* όλα τα βήματα-κλήσεις συναρτήσεων και στα ονόματα συναρτήσεων-μελών να βάζεις και το όνομα της κλάσης –π.χ. όχι “`appendFuncName("load")`” αλλά πιο συγκεκριμένα “`appendFuncName( "Course::load")`”– το “`100`” μπορεί να μην είναι αρκετό.

Ένα δεύτερο παράδειγμα χρήσης αυτών των εργαλείων μπορούμε να δούμε στον δημιουργό της *DateTime* όπως τον ξαναγράψαμε στην §24.5.

Αλλάζουμε την κλάση εξαιρέσεων:

```
struct DateXptn :  public exception
{
   static const int fnLength = 100;
   enum { yearErr, monthErr, dayErr, outOfLimits,
          fileNotOpen, cannotRead, cannotWrite };
   char funcName[ fnLength ];
   int  errorCode;
   int  errVal1;
   int  errVal2;
   int  errVal3;
```

```
   Date errDateVal;
   DateXptn( const char* fn, int ec,
             int ev1 = 0, int ev2 = 0, int ev3 = 0 )
      : errorCode(ec), errVal1(ev1), errVal2(ev2), errVal3(ev3)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   DateXptn( const char* fn, int ec, int ev1, const Date& ed )
      : errorCode(ec), errVal1(ev1), errDateVal(ed)
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';  }
   virtual ~DateXptn() throw() {  };

   void appendFuncName( const char* fn )
   // ΟΠΩΣ ΣΤΗ BStringXptn

   void displayFuncName( ostream& tout )
   // ΟΠΩΣ ΣΤΗ BStringXptn
}; // DateXptn
```

και τον δημιουργό της *DateTime*:

```
DateTime::DateTime( int yp, int mp, int dp, int hp, int minp, int sp )
try   :  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;
}
catch( DateTimeXptn& x )
{  throw;  }
catch( DateXptn& x )
{
   x.appendFuncName( "DateTime" );
   throw;
} // DateTime::DateTime
```

Τώρα μπορεί να έχουμε ένα μήνυμα σαν

```
day error ( 44 ) thrown in Date
 called by DateTime
```

που είναι ακριβέστερο από το να πούμε απλώς ότι «*βρήκαμε λάθος στον δημιουργό της DateTime.*»

Ας δούμε τώρα πώς θα αξιοποιήσουμε αυτά τα εργαλεία. Μήπως θα πρέπει να ξαναγράψουμε όλες τις συναρτήσεις και όλες τις μεθόδους ώστε να πιάνουμε ολόκληρη την ακολουθία κλήσεων; Ούτε να το σκέφτεσαι!

- ***Η διαδικασία «πιάσε – διόρθωσε – ξαναρίξε» είναι χρονοβόρα.***

Δεν μπορείς επομένως να την έχεις παντού σε ένα πρόγραμμα που είναι σε εκμετάλλευση. Και αν μεν η ρίψη της εξαίρεσης θα έχει αποτέλεσμα την διακοπή της εκτέλεσης του προγράμματος οι καθυστερήσεις δεν μας πειράζουν. Αν όμως υπάρχει δυνατότητα να κάνεις διόρθωση και να συνεχίσεις την εκτέλεση το πράγμα αλλάζει.

Όταν κάνεις την ανάπτυξη του προγράμματος και ψάχνεις για να εντοπίσεις και να διορθώσεις μερικά –ενδεχομένως δύσκολα– λάθη αυτά τα εργαλεία μπορεί να σε βοηθήσουν.

Μπορεί να τα χρησιμοποιήσεις και στο πρόγραμμα που είναι σε εκμετάλλευση αλλά σε σημεία που θα επιλέξεις με προσοχή.

Σε κάθε περίπτωση πάρε υπόψη σου και το εξής: *Υπάρχουν ακολουθίες κλήσεων που ταυτοποιούνται με μοναδικό τρόπο από την αρχή και το τέλος.*

24.9 Ασφάλεια ως προς τις Εξαιρέσεις

Στις §21.6 και §21.6.1 είδαμε δύο μορφές της `BString::operator=()` που είπαμε ότι έχουν **ασφάλεια προς τις εξαιρέσεις** (exception safety) επιπέδου **ισχυρής εγγύησης** (strong guarantee). Όπως είπαμε, με τον όρο αυτόν εννοούμε ότι αυτή η μέθοδος (επιφόρτωσης) έχει τα εξής χαρακτηριστικά: σε περίπτωση που θα εγερθεί εξαίρεση όταν εκτελείται η "`a = b`"

- Δεν θα έχουμε διαρροή μνήμης.
- Δεν θα αλλάξει η τιμή της *a*.

Η ασφάλεια ισχυρής εγγύησης της δεύτερης μορφής στηρίζεται στη σιγουριά μας ότι η *BString::swap()* δεν θα ρίξει εξαίρεση. Λέμε ότι η *BString::swap()* έχει **ασφάλεια προς τις εξαιρέσεις** σε επίπεδο **εγγύησης μη-ρίψης** (no-throw guarantee).

Στην §20.7.2.3 είδαμε την

```
void Route::erase1RouteStop( int ndx )
{
   for ( int k(ndx); k <= rNoOfStops; ++k )
      rAllStops[k] = rAllStops[k+1];
   --rNoOfStops;
} // Route::erase1RouteStop
```

που «σβήνει» το στοιχείο που βρίσκεται στη θέση *ndx* σε έναν ταξινομημένο πίνακα. Λέγαμε όμως ότι κάποια από τις "`rAllStops[k] = rAllStops[k+1]`" της *erase1RouteStop()* μπορεί να ρίξει *bad_alloc* διότι τα `rAllStops[k]`, που είναι κλάσης *RouteStop*, έχουν μέλος "`string sName`".

Ας πούμε λοιπόν ότι ρίχνεται η εξαίρεση όταν ο δείκτης *k* έχει τιμή k_0. Ο τελεστής εκχώρησης της *string* έχει ασφάλεια ισχυρής εγγύησης που σημαίνει:

- δεν έχουμε διαρροή μνήμης και
- το `rAllStops[`k_0`]` κρατάει την αρχική του τιμή. Επομένως:
  - Αν k_0 == *ndx* σημαίνει ότι ο πίνακας έχει μείνει αθικτος και η αναλλοίωτη της κλάσης ισχύει.
  - Αν k_0 > *ndx*, στο προηγούμενο βήμα της **for** το `rAllStops[`k_0`-1]` έχει πάρει την τιμή του `rAllStops[`k_0`]`. Έτσι, ενώ έχουμε σβήσει το `rAllStops[`*ndx*`]`, το πλήθος των στοιχείων μεταξύ των δύο φρουρών παραμένει αυτό που ήταν αφού τα στοιχεία στις θέσεις k_0-1 και k_0 έχουν την ίδια τιμή. Αυτή η κατάσταση παραβιάζει την αναλλοίωτη της κλάσης και συγκεκριμένα το κομμάτι:

$$(\forall\, k: 1..rNoOfStops-1 \bullet (\text{rAllStops}[k+1].sDist > \text{rAllStops}[k].sDist)$$

Πάντως: και στις δύο περιπτώσεις το βέλος `rAllStops` δείχνει τον δυναμικό πίνακα, το *rReserved* μας λέει πόση δυναμική μνήμη έχουμε πάρει και το *rNoOfStops* πόση από αυτήν είναι σε χρήση. Δηλαδή το αντικείμενο είναι διαχειρίσιμο είτε με τον καταστροφέα είτε με την *Route::clearRouteStops()*.

Λέμε ότι η *Route::erase1RouteStop()* έχει **ασφάλεια** επιπέδου **βασικής εγγύησης** (basic guarantee).[22]

Να τα τρία επίπεδα εγγύησης ασφάλειας μιας μεθόδου (ή μιας συνάρτησης που διαχειρίζεται ένα αντικείμενο) ως προς τις εξαιρέσεις (Abrahams 2001):

- Η *βασική εγγύηση*: *Δεν υπάρχει διαρροή πόρων και η αναλλοίωτη του αντικειμένου διατηρείται.* Το ότι ισχύει η αναλλοίωτη σημαίνει ότι μπορεί να κληθεί ο καταστροφέας (ας πούμε με το ξετύλιγμα της στοίβας) που έχει την αναλλοίωτη ως προϋπόθεση. Κατά τα άλλα, αυτό δεν λέει και πολλά πράγματα διότι καταστάσεις που να ισχύει η αναλλοίωτη μπορεί να υπάρχουν πολλές. Αν θέλεις να συνεχίσεις να δουλεύεις με το αντικείμε-

[22] Μερικοί ονομάζουν αυτό το επίπεδο **ελάχιστης** (minimal) εγγύησης διότι δεν ισχύει ολόκληρη η αναλλοίωτη.

νο θα πρέπει να το φέρεις σε συγκεκριμένη κατάσταση π.χ. να κάνεις κάποιο είδος επανεκκίνησης.

- Η *ισχυρή* εγγύηση: Η εκτέλεση της μεθόδου
  - *είτε ολοκληρώνεται επιτυχώς*
  - *είτε διακόπτεται με έγερση εξαίρεσης αλλά το πρόγραμμα* (και το αντικείμενο) *βρίσκεται στην κατάσταση που ήταν πριν αρχίσει η εκτέλεση της μεθόδου. Δεν υπάρχει διαρροή πόρων.*
- Η εγγύηση *μη-ρίψης*: *Η εκτέλεση της μεθόδου ολοκληρώνεται επιτυχώς χωρίς να ριχτεί (ή να περάσει μέσα από αυτήν) εξαίρεση.*

Προφανώς για κάθε μέθοδο θα θέλαμε να έχουμε το μέγιστο δυνατό επίπεδο ασφάλειας. Ποιο ακριβώς είναι το επίπεδο εξαρτάται από τη συγκεκριμένη μέθοδο. Και πώς πετυχαίνουμε το επίπεδο που θέλουμε; Ας ξαναγυρίσουμε, για παράδειγμα, στη *Route::erase1-RouteStop* και ας πούμε ότι προσπαθούμε να την κάνουμε να έχει ισχυρή εγγύηση ασφάλειας. Μπορούμε να ξαναδώσουμε την αρχική τιμή στα στοιχεία που αλλάξαμε; Ούτε λόγος! Θα πρέπει να κάνουμε αντιγραφές που –λογικώς– θα ρίξουν εξαίρεση. Ο μόνος τρόπος είναι ο εξής: Να μην πειράξουμε τον αρχικό πίνακα αλλά να τον αντιγράψουμε σε έναν άλλον

```
RouteStop* tmp( new RouteStop[rReserved] );
```

παραλείποντας το `rAllStops[`*ndx*`]`:

- Αν η αντιγραφή ολοκληρωθεί επιτυχώς ανταλλάσσουμε τις τιμές των βελών *rAllStops* και *tmp*.
- Αλλιώς, αν ριχτεί εξαίρεση, ο πίνακας παραμένει όπως ήταν.

Δηλαδή: για να διαγράψουμε μια τιμή αντιγράφουμε ολόκληρον τον πίνακα. Είναι δεκτό αυτό; Αλλά και τώρα μήπως κάνουμε κάτι καλύτερο; Κατά μέσον όρο αντιγράφουμε τον μισό πίνακα! Γενικώς η αντιγραφή πίνακα –για διαγραφή ή εισαγωγή μιας τιμής– φαίνεται απαράδεκτη αλλά δεν έχουμε και άλλη επιλογή (πάντως εδώ οι πίνακες δεν είναι και τόσο μεγάλοι).

Δηλαδή το καλό επίπεδο ασφάλειας για τις εξαιρέσεις πληρώνεται με (μεγάλο) υπολογιστικό χρόνο; Όχι απαραιτήτως. Η λύση στο πρόβλημά μας βρίσκεται στην αλλαγή δομής δεδομένων για τις στάσεις που αποτελούν ένα σύνολο: όπως θα δεις αργότερα, χρησιμοποιώντας τα εργαλεία που μας δίνει η STL, θα τις βάλουμε σε ένα σύνολο με στοιχεία τύπου *RouteStop* ή, όπως θα το γράφουμε, "`set<RouteStop>`"· σε αυτή τη δομή οι εισαγωγές και οι διαγραφές στοιχείων γίνονται με ισχυρή εγγύηση ασφάλειας χωρίς να είναι χρονοβόρες.

Επομένως

- ♦ ***Για να έχουμε το επιθυμητό επίπεδο εγγύησης ασφάλειας πρέπει να σχεδιάσουμε καταλλήλως το πρόγραμμά μας.***

Για να διασφαλίσουμε τη μη-διαρροή πόρων χρησιμοποιούμε τις **try**/**catch**. Η τεχνική RAII είναι σαφώς καλύτερη και ασφαλέστερη. Με τα εργαλεία της STL, που θα μάθουμε αργότερα, η εφαρμογή της θα γίνει απλούστερη.

Ένα εργαλείο, πολύ συχνά χρήσιμο, είναι η (ασφαλής) *swap()*. Μερικές φορές θα πρέπει να συνδυάσεις τη χρήση της με την τεχνική PIMPL (§22.10).

24.10 Σύνοψη

Η C μας δίνει τη δυνατότητα να βρούμε τί δεν πήγε καλά όταν καλέσαμε μια από τις συναρτήσεις της βιβλιοθήκης της

- από την τιμή που επιστρέφει η συνάρτηση και

- την τιμή της καθολικής μεταβλητής *errno*.

Μας δίνει τη δυνατότητα να διακόψουμε την εκτέλεση του προγράμματος (*abort()*, *exit()*, *_Exit()*) και να καθορίσουμε τι θα συμβεί στην περίπτωση αυτή (*atexit()*).

Με την *assert()* μπορούμε να ελέγχουμε τις συνθήκες επαλήθευσης στα διάφορα σημεία εκτέλεσης του προγράμματος. Αλλά, όπως ξέρουμε, αν η συνθήκη δεν ισχύει η «ποινή είναι βαριά».

Για τις εξαιρέσεις της C++ μάθαμε ότι

- Αν η εξαίρεση τελειώσει το «ταξίδι» της χωρίς να συλληφθεί τότε καλείται η *terminate()* (που καλεί την *abort()*) για να διακόψει την εκτέλεση του προγράμματος.
- Αν παραβιασθεί μια προδιαγραφή εξαιρέσεων –δηλαδή αν από μια συνάρτηση ριχτεί εξαίρεση που ο τύπος της δεν υπάρχει στην προδιαγραφή εξαιρέσεων– καλείται η συνάρτηση *unexpected()*, που με τη σειρά της θα καλέσει την *terminate()*.

Οι *set_terminate()* και *set_unexpected()* σου επιτρέπουν να αντικαταστήσεις τις *terminate()* και *unexpected()* αντιστοίχως.

Αν έλθουμε στο «πρακτέο» θα πρέπει να πούμε κατ' αρχάς ότι καλό είναι να αφήσεις κατά μέρος τις προδιαγραφές εξαιρέσεων εκτός από την "`throw()`".

Αν γράφεις δικές σου κλάσεις εξαιρέσεων (ή χρησιμοποιείς τις δικές μας) καλό είναι να τις βάζεις να κληρονομούν (αμέσως ή εμμέσως) την *exception*.

Όταν γράφεις το πρόγραμμά σου στόχος σου θα πρέπει να είναι η καλύτερη δυνατή εγγύηση ασφάλειας ως προς τις εξαιρέσεις. Η τεχνική RAII και η ασφαλής *swap()* είναι εργαλεία που μπορείς να χρησιμοποιήσεις.

Πάντως

- η ασφάλεια ως προς τις εξαιρέσεις και
- το πού θα πιάνεις και πώς θα χειρίζεσαι τις εξαιρέσεις

έχουν σχέση (και) με τη σχεδίαση του προγράμματός σου.