κεφάλαιο

18

Προετοιμάζοντας Βιβλιοθήκες

Ο στόχος μας σε αυτό το κεφάλαιο:
Να μάθεις να κάνεις (στατικές) βιβλιοθήκες συναρτήσεων

- είτε διότι το ζήτησε κάποιος πελάτης (ή εργοδότης)
- είτε για να κεφαλαιοποιήσεις την προγραμματιστική δουλειά που κάνεις, ώστε να μη χρειάζεται να ανακαλύπτεις τον τροχό κάθε τόσο.

Προσδοκώμενα αποτελέσματα:
Θα μπορείς να γράψεις και να χρησιμοποιήσεις στατικές βιβλιοθήκες συναρτήσεων και περιγραμμάτων.

Έννοιες κλειδιά:

- *οδηγία* `define`
- *οδηγία* `ifdef`
- *βιβλιοθήκη περιγραμμάτων συναρτήσεων*
- *στατική βιβλιοθήκη*
- *χωριστή μεταγλώττιση*
- `namespace`

Περιεχόμενα:

18.1 Οι Οδηγίες "`define`", "`ifdef`" κλπ 582
18.2 Μια Βιβλιοθήκη Περιγραμμάτων Συναρτήσεων 586
18.3 Χωριστή Μεταγλώττιση 588
18.4 Μια Στατική Βιβλιοθήκη 592
18.5 "`namespace`": Το Πρόβλημα και η Λύση 593
18.6 Ανακεφαλαίωση 596
Ασκήσεις 597
Α Ομάδα 597
Β Ομάδα 597

Εισαγωγικές Παρατηρήσεις:
Αρχίζοντας, να τονίσουμε ότι οι προγραμματιστικές βιβλιοθήκες με τις οποίες θα ασχοληθούμε είναι *βιβλιοθήκες συναρτήσεων* και όχι εκτελέσιμων προγραμμάτων (υπάρχουν και τέτοιες). Οι πρώτες εμφανίστηκαν σχετικώς νωρίς με πιο γνωστές –τα πρώτα χρόνια– τις βιβλιοθήκες της FORTRAN.

Οι συναρτήσεις των βιβλιοθηκών *συνδέονται* στα προγράμματα που αναπτύσσουν οι προγραμματιστές απαλλάσσοντάς τους έτσι από το να «ξαναεφευρίσκουν τον τροχό»!

Οι προγραμματιστές αναπτύσσουν βιβλιοθήκες συναρτήσεων «κατά παραγγελία». Αλλά πολλές βιβλιοθήκες προκύπτουν και ως παραπροϊόντα της ανάπτυξης προγραμμάτων: ο προγραμματιστής, κρίνοντας από την εμπειρία του, ότι κάποιες συναρτήσεις έχουν γενι-

κότερη χρησιμότητα και όχι μόνο για το πρόγραμμα που γράφηκαν τις εντάσσει σε βιβλιοθήκες ώστε να μπορεί να τις ξαναχρησιμοποιήσει.

Μια βιβλιοθήκη, σε σχέση με τον τρόπο που χειρίζεται τις συνιστώσες της ο **συνδέτης** (linker), μπορεί να είναι **στατική** (static) ή **δυναμικής σύνδεσης** (dynamic linking):

- *Στατικές Βιβλιοθήκες*: Μετά τη μεταγλώττιση του (αρχικού) προγράμματος ο συνδέτης εντάσσει σε αυτό τις συναρτήσεις που καλούνται από τη βιβλιοθήκη δημιουργώντας το τελικό εκτελέσιμο πρόγραμμα που είναι αυθύπαρκτο, μπορεί δηλαδή να εκτελεσθεί χωρίς την παρουσία της βιβλιοθήκης.
- *Βιβλιοθήκες Δυναμικής Σύνδεσης*: Τέτοιες είναι οι *DLL* (Dynamic Link Library) των Windows και οι *DSO* (Dynamic Shared Object) των Unix, Linux. Στη φάση της δημιουργίας του εκτελέσιμου προγράμματος ο συνδέτης δεν αντιγράφει σε αυτό τις συναρτήσεις που καλούνται· απλώς καταγράφει τι καλείται και σε ποιο σημείο. *Η τελική φάση της σύνδεσης γίνεται κάθε φορά που αρχίζει η εκτέλεση του προγράμματος*: ο συνδέτης αναζητεί τις δυναμικές βιβλιοθήκες, τις φορτώνει στη μνήμη και κάνει την τελική σύνδεση. *Αν κάποια βιβλιοθήκη είναι ήδη φορτωμένη, επειδή τη ζήτησε άλλο πρόγραμμα, δεν ξαναφορτώνεται.* Η δυναμική βιβλιοθήκη σβήνεται από τη μνήμη όταν δεν εκτελείται πρόγραμμα που να έχει συνδεθεί με αυτήν. Ένα (εκτελέσιμο) πρόγραμμα που χρησιμοποιεί βιβλιοθήκες δυναμικής σύνδεσης *δεν είναι αυθύπαρκτο.* Όταν το μεταφέρεις από τη μια εγκατάσταση στην άλλη θα πρέπει να φροντίσεις να υπάρχουν –στη νέα εγκατάσταση– και οι απαιτούμενες βιβλιοθήκες.

Πέρα από τις βιβλιοθήκες συναρτήσεων υπάρχουν βιβλιοθήκες περιγραμμάτων (συναρτήσεων ή κλάσεων), κλάσεων κλπ.

Εδώ θα ασχοληθούμε με

- στατικές βιβλιοθήκες συναρτήσεων και
- βιβλιοθήκες περιγραμμάτων συναρτήσεων.

Πριν ξεκινήσουμε όμως θα πρέπει να δούμε μερικές «οδηγίες προς τον προεπεξεργαστή» που θα μας είναι απαραίτητες από εδώ και πέρα: τις **define** και **undef** καθώς και τις **ifdef** και **ifndef**.

> *ΠΡΟΣΟΧΗ! Σε αυτό το κεφάλαιο, για να δείξουμε ορισμένα πράγματα, θα πρέπει να χρησιμοποιήσουμε διαδικασίες που εξαρτώνται από το περιβάλλον ανάπτυξης και το ΛΣ. Εδώ θα στηριχτούμε στον Dev-C++, σε περιβάλλον Windows. Εσύ θα πρέπει, πιθανότατα, να ψάξεις τα εγχειρίδια (ή το Help) της C++ που χρησιμοποιείς.*

18.1 Οι Οδηγίες “`define`”, “`ifdef`” κλπ

Στην §1.5 είδαμε την οδηγία προς τον προεπεξεργαστή “`include`” και από τότε τη χρησιμοποιούμε σε όλα τα προγράμματά μας.

Στην επόμενη παράγαρφο –αλλά και στα επόμενα κεφάλαια– θα χρειαστούμε μερικές οδηγίες ακόμη· τις βλέπουμε στη παρούσα παράγραφο.

Η οδηγία “`define`” (όρισε) έχει συντακτικό:

"`#`", "`define`", *όνομα*, *λίστα λεξικών οντοτήτων*

και ορίζει μια **μακροεντολή** ή **μακροσυνάρτηση** (macro). Το *όνομα* είναι το **όνομα της μακροεντολής** (macro identifier), ενώ η *λίστα λεξικών οντοτήτων* είναι το **σώμα της μακροεντολής** (macro body).

Αποτέλεσμα της οδηγίας είναι να αντικατασταθεί, στις γραμμές που ακολουθούν, το *όνομα* της μακροεντολής από αυτά που ορίζει η *λίστα λεξικών οντοτήτων*. Αυτή η διαδικασία λέγεται **μακροανάπτυξη** (macro expansion).

Η οδηγία “`define`” αναιρείται από μια οδηγία “`undef`”(ine):

"#", "undef", *όνομα*
με το ίδιο όνομα μακροεντολής.

Μετά από οδηγία "`undef`" το όνομα της μακροεντολής είναι **αόριστο** (undefined), όπως και πριν από την οδηγία "`define`". Μετά την define είναι **ορισμένο** (defined).

Παράδειγμα

Η οδηγία:

```
#define NMAX 50
```

ζητάει να αντικατασταθεί η λέξη *NMAX* με το "`50`" οπουδήποτε βρεθεί στη συνέχεια μέχρι να βρεθεί οδηγία "`#undef NMAX`".

Στο παρακάτω κομμάτι προγράμματος:

```
#define NMAX 50

int main()
{
   int a[NMAX];
      :
   for (int k(0); k <= NMAX-1; ++k)
   {
#undef NMAX
       swap(a[k], a[NMAX-k]);
   } // for
      :
```

η αντικατάσταση της *NMAX* από την τιμή 50 γίνεται μέχρι και τη γραμμή με την εντολή **for** ενώ δεν γίνεται στη γραμμή με την εντολή `swap(a[k], a[NMAX-k])` διότι προηγουμένως μεσολαβεί η οδηγία "`#undef NMAX`".Έτσι, ο μεταγλωττιστής θα μας βγάλει το μήνυμα "`Undefined symbol 'NMAX'`" για τη γραμμή αυτή.

Το *NMAX* είναι *ορισμένο* στις γραμμές του προγράμματος μεταξύ των οδηγιών "`#define NMAX 50`" και "`#undef NMAX`" ενώ είναι *αόριστο* πριν από την πρώτη και μετά τη δεύτερη.

Οι αντικαταστάσεις μπορεί να είναι και πιο πολύπλοκες:

Παράδειγμα

Αντιγράφουμε από το `stdlib.h`:

```
#define max(a,b)    (((a) > (b)) ? (a) : (b))
```

Αυτή η οδηγία θα έχει ως αποτέλεσμα μια εντολή σαν την:

```
      y = x + max(x+1, y-1);
```

να γίνει:

```
      y = x + (((x+1) > (y-1)) ? (x+1) : (y-1));
```

Η μακροανάπτυξη *δεν γίνεται* όταν το όνομα της μακροεντολής ευρεθεί:

- μέσα σε ορμαθό χαρακτήρων ή σε σχόλιο,
- μέσα στην ίδια τη μακροεντολή (π.χ. η "`#define A A`" δεν θα έχει ως αποτέλεσμα την επ' άπειρο εκτέλεσή της)

Φυσικά, μακροανάπτυξη *δεν γίνεται* και μέσα σε μια οδηγία "`undef`". Δηλαδή, για παράδειγμα, η

```
#define NMAX 50
```

δεν θα κάνει την

```
#undef NMAX
```

που ακολουθεί "`#undef 50`".

Και τι κερδίσαμε; Απλές συναρτήσεις, που καλούνται πολύ συχνά δεν «φορτώνουν» τον χρόνο εκτέλεσης του προγράμματος με το κόστος κλήσης συνάρτησης. Αυτό το πρόβλημα όμως το αντιμετωπίσαμε ήδη στην §14.1 με τις συναρτήσεις "`inline`". Έτσι, όσα είπαμε μέχρι εδώ για τη "`define`" σκοπό έχουν να την αναγνωρίζεις και να καταλαβαίνεις τη χρήση της σε άλλα προγράμματα. Στη C++

- η δήλωση σταθερών με "`const`" και
- οι συναρτήσεις "`inline`"

λύνουν τα αντίστοιχα προβλήματα με πολύ καλύτερο τρόπο και αυτά τα εργαλεία θα χρησιμοποιούμε.

Τώρα θα δούμε μια «περίεργη» μορφή της "`define`" την οποία και θα χρησιμοποιήσουμε στη συνέχεια.

Αν δεν υπάρχει σώμα της μακροεντολής τότε ο προεπεξεργαστής θα διαγράψει το όνομά της όπου το βρει. Π.χ. το:

```
#define NMAX
int main()
{
   int a[NMAX];

   for (int k(0); k <= NMAX-1; ++k)
   {
      swap(a[k], a[NMAX-k]);
   }
```

θα γίνει:

```
#define NMAX
int main()
{
   int a[];

   for (int k(0); k <= -1; ++k)
   {
      swap(a[k], a[-k]);
   }
```

Μια τέτοια χρήση της "`define`" κάνουμε όταν θέλουμε να έχουμε *ορισμένο* κάποιο όνομα σε μια περιοχή του προγράμματός μας, όπως θα δούμε στη συνέχεια.

Πολύ συχνά, θέλουμε να μεταγλωττίσουμε (ή να μη μεταγλωττίσουμε) ένα κομμάτι προγράμματος με κριτήριο το αν έχει ορισθεί ή δεν έχει ορισθεί κάποιο σύμβολο χωρίς να μας ενδιαφέρει η τιμή που μπορεί να του δόθηκε.

Ας πούμε ότι, σε κάποιο πρόγραμμα, αν έχει ορισθεί το σύμβολο N θέλουμε να έχουμε ως συνάρτηση f την

```
int f( int x )
{  ++x;  return x;  }
```

αλλιώς, αν δεν έχει ορισθεί το N, θέλουμε ως f την

```
int f( int x )
{  return x;  }
```

Αυτό γράφεται ως εξής:[1]

```
#ifdef N

int f( int x )
{  ++x;  return x;  }

#endif
```

[1] Υπάρχει και οδηγία "`#else`" και θα μπορούσαμε να το γράψουμε διαφορετικά αλλά ας την αφήσουμε...

```
#ifndef N

int f( int x )
{  return x;  }

#endif
```

Η "`#ifdef N`" είναι ο τρόπος που μας δίνει η C++ για να διατυπώσουμε την οδηγία: «`#if έχει ορισθεί το N`».

Για να διατυπώσουμε την "`#if δεν έχει ορισθεί το N`" θα πρέπει να γράψουμε: "`#ifndef N`".

Η "`#endif`" σημειώνει το τέλος αυτών που ισχύουν για την αντίστοιχη **if** ("`#ifdef N` " ή "`#ifndef N` ").

Παράδειγμα ✎

Αντιγράφουμε από το `iostream` (gcc – Dev-C++):

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

// . . . όλο το περιεχόμενο του αρχείου

#endif /* _GLIBCXX_IOSTREAM */
```

Ας δούμε πρώτα το πρόβλημα και μετά το νόημα των παραπάνω. Ας πούμε ότι έχεις ένα αρχείο, το `myFuncs.h`, με κάποιες συναρτήσεις. Μέσα στο αρχείο έχεις βάλει τη οδηγία:

```
#include <iostream>
```

Στο αρχείο `myprog.cpp`, που περιέχει ένα πρόγραμμα που γράφεις, βάζεις στην αρχή:

```
#include <iostream>
#include "myFuncs.h"
```

Τι θα μπορούσε να συμβεί στην περίπτωση αυτή; Οι δηλώσεις και οι ορισμοί του `iostream` θα υπάρχουν στο πρόγραμμά σου δύο φορές και αυτό δεν θα γίνει δεκτό από τον μεταγλωττιστή.

Το `iostream` αμύνεται, με τις οδηγίες που είδαμε πιο πάνω, ως εξής:

- Όταν εκτελεσθεί η "`#include <iostream>`" του `myprog.cpp` το σύμβολο "`_GLIBCXX_IOSTREAM`" δεν είναι ορισμένο. Έτσι, λόγω της
  `#ifndef _GLIBCXX_IOSTREAM`
  περικλείεται στο πρόγραμμά μας ολόκληρο το περιεχόμενο του `iostream`. Πρώτη και καλύτερη περικλείεται και εκτελείται η οδηγία
  `#define _GLIBCXX_IOSTREAM 1`
  Τώρα πια το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο.[2]
- Στη συνέχεια, όταν περιληφθεί το `myFuncs.h`, λόγω της οδηγίας
  `#include <iostream>`
  που υπάρχει εκεί, περιλαμβάνεται για δεύτερη φορά το `iostream`. Τώρα όμως, όταν εκτελείται η
  `#ifndef _GLIBCXX_IOSTREAM`
  το `_GLIBCXX_IOSTREAM` *είναι* ορισμένο και δεν περιλαμβάνεται οτιδήποτε υπάρχει μέχρι και την
  `#endif /* _GLIBCXX_IOSTREAM */`

Αυτό θα πρέπει να κάνουμε και εμείς στα αρχεία με ορισμούς και δηλώσεις που θέλουμε να χρησιμοποιούμε συχνά. Για παράδειγμα, στο `myFuncs.h` θα πρέπει να περιλάβουμε όλο το πριεχόμενο μεταξύ των οδηγιών:

```
#ifndef _MYFUNCS_H
#define _MYFUNCS_H
```

[2] Εκείνο το "1" στη **define** δεν ειναι απαραίτητο, κατ' αρχήν.

```
// . . . όλο το περιεχόμενο του αρχείου

#endif  /* _MYFUNCS_H */
```
❧❧❧

18.2 Μια Βιβλιοθήκη Περιγραμμάτων Συναρτήσεων

Το πιο απλό είδος βιβλιοθήκης είναι μια βιβλιοθήκη περιγραμμάτων συναρτήσεων. Θα κάνουμε λοιπόν και εμείς μια μικρή βιβλιοθήκη –τη *MyTmpltLib*– που θα περιέχει περιγράμματα που ήδη έχουμε χρησιμοποιήσει:

- Το περίγραμμα συνάρτησης *linSearch()*, που είδαμε στην §16.13.1, παρ' όλο που είπαμε ότι στις βιβλιοθήκες της C++ υπάρχει η *find()*.
- Το *renew()*, (§16.12), που μας είναι χρήσιμο μέχρι να μάθουμε τα περιγράμματα περιεχόντων (containers) της C++.
- Τα *new2d()* και *delete2d()* που είδαμε στην §16.9.

Σε ένα αρχείο, ας το πούμε `MyTmpltLib.h`, αντιγράφουμε τα τέσσερα περιγράμματα και πριν από όλα την κλάση εξαιρέσεων *MyTmpltLibXptn* (§16.9):

```
#ifndef _MYTMPLLIB_H
#define _MYTMPLLIB_H

#include <string>
#include <new>

using namespace std;

struct MyTmpltLibXptn
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// ΟΠΩΣ ΣΤΗΝ §16.13.1

template< typename T >
void renew( T*& p, int ni, int nf )
// ΟΠΩΣ ΣΤΗΝ §16.12

template< typename T >
T** new2d( int nR, int nC )
// ΟΠΩΣ ΣΤΗΝ §16.9

template< typename T >
void delete2d( T**& a, int nR )
// ΟΠΩΣ ΣΤΗΝ §16.9

#endif // _MYTMPLLIB_H
```

Τα πάντα περιέχονται ανάμεσα στις `#ifndef _MYTMPLLIB_H` και `#endif`. Πριν από όλα η `#define _MYTMPLLIB_H`. Με αυτές αμυνόμαστε για περίπτωση πολλαπλής `#include "MyTmpltLib.h"`.

Ακόμη:

- Η `#include <string>` μας είναι απαραίτητη για τη *strncpy()*.
- Η `#include <new>` μας είναι απαραίτητη για τη *bad_alloc*.
- Η using `namespace std` μας είναι απαραίτητη για να μη γράφουμε `std::strncpy`, `std::bad_alloc`.

Να δούμε τώρα πώς τη χρησιμοποιούμε. Το πρόγραμμα της §16.12 γίνεται:

```
#include <iostream>
```

```
#include <new>
#include "MyTmpltLib.h"

using namespace std;

int main()
{
   int*    ip;
   double* dp;
// . . .
   renew( ip, 3, 7 );  renew( dp, 3, 7 );
// . . .
} // main
```

Δηλαδή: υπάρχει η `#include "MyTmpltLib.h"` και δεν υπάρχει το περίγραμμα *renew()*. Από ολόκληρο το `MyTmpltLib.h` χρησιμοποιείται μόνο το *renew* από το οποίο δημιουργούνται δύο περιπτώσεις: μια για `int` (λογω της `renew(ip, 3, 7)`) και μια για `double` (λογω της `renew(dp, 3, 7)`).

Στο πρόγραμμα του Παράδ. 2 της §16.13 θα έχουμε:

```
#include <iostream>
#include <fstream>
#include <string>
#include <new>
#include "MyTmpltLib.h"

using namespace std;

struct ApplicXptn
// . . .
```

Και εδώ βάλαμε την `#include "MyTmpltLib.h"` και αφαιρέσαμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *renew()* και *linSearch()*. Από το *renew()* θα πάρουμε ένα στιγμιότυπο για τον τύπο *GrElmn* και από τη *linSearch()* ένα στιγμιότυπο για τον ίδιο τύπο.

Στο πρόγραμμα πολλαπλασιαμού πινάκων (§16.9) θα έχουμε:

```
#include <iostream>
#include <fstream>
#include "MyTmpltLib.h"

using namespace std;

void input2DAr( istream& tin, int** a, int nRow, int nCol );
void output2DAr( ostream& tout, int** a, int nRow, int nCol );

int main()
// . . .
```

Και πάλι βάζουμε την `#include "MyTmpltLib.h"` και αφαιρούμε την κλάση εξαιρέσεων *MyTmpltLibXptn* και τα περιγράμματα *new2d* και *delete2d*. Παίρνουμε μια περίπτωση για το κάθε ένα για τον τύπο `int`.

Ως προς τη σύνδεση: τι είδους βιβλιοθήκη κάναμε; Στατική ή δυναμικής σύνδεσης; Παρ' όλο που οι όροι αυτοί αναφέρονται σε βιβλιοθήκες με μεταγλωττισμένο περιεχόμενο, μπορούμε να πούμε ότι έχουμε μια βιβλιοθήκη *στατική*· ο συνδέτης καλείται μόνο μια φορά να δώσει ένα αυθύπαρκτο εκτελέσιμο πρόγραμμα.

Όπως βλέπεις, τα πράγματα είναι πολύ απλά. Βέβαια, αν δεις τα πράγματα από εμπορική άποψη υπάρχει ένα προβληματάκι: Αν θέλεις να πουλήσεις προγράμματα σε κάποιον πελάτη του δίνεις υποχρεωτικώς το αρχικό πρόγραμμα, σε C++. Αλλά το αρχικό πρόγραμμα, συνήθως, τιμολογείται πολύ ακριβά. Είναι δυνατόν να του δίνεις μεταγλωττισμένες συναρτήσεις μόνο; Αυτό βλέπουμε στη συνέχεια.

18.3 Χωριστή Μεταγλώττιση

Ο επόμενος στόχος μας είναι να δημιουργήσουμε μια ακόμη πιο μικρή βιβλιοθήκη με τους δύο αριθμητικούς αλγόριθμους που έχουμε μεταφράσει σε πρόγραμμα: τον *αλγόριθμο του Horner* (§9.4, Παράδ. 1) και τον αλγόριθμο της διχοτόμησης (bisection, §17.14).

Ας ξεκινήσουμε ως εξής: Βάζουμε τις δύο συναρτήσεις σε ένα αρχείο με όνομα `MyNumerics.cpp`. Στο αρχείο `MyNumerics.h` βάζουμε τα εξής:

```
#ifndef _MYNUMERICS_H
#define _MYNUMERICS_H

#include <string>

using namespace std;

struct MyNumericsXptn
{
   enum {  domainError, noArray  };

   char   funcName[100];
   int    errCode;
   double errDblVal;
   MyNumericsXptn( const char* fn, int ec, double ev = 0 )
   {  strncpy( funcName, fn, 99 );  funcName[99] = '\0';
      errCode = ec;  errDblVal = ev;  }
}; // MyNumericsXptn

// ph -- Υπολογισμός τιμής πολυωνύμου, βαθμού m, με τον
//       αλγόριθμο του Horner.
double ph( const double a[], int m, double x );

// bisection -- προσεγγίζει λύση (root) της εξίσωσης f(x) = 0
//              στο διάστημα [α,β] με τη μέθοδο της διχοτόμησης
//              με nMax επαναλήψεις το πολύ.
//              Στο [a,b] η f πρέπει να είναι συνεχής και
//              f(a)*f(b) <= 0
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );

#endif // _MYNUMERICS_H
```

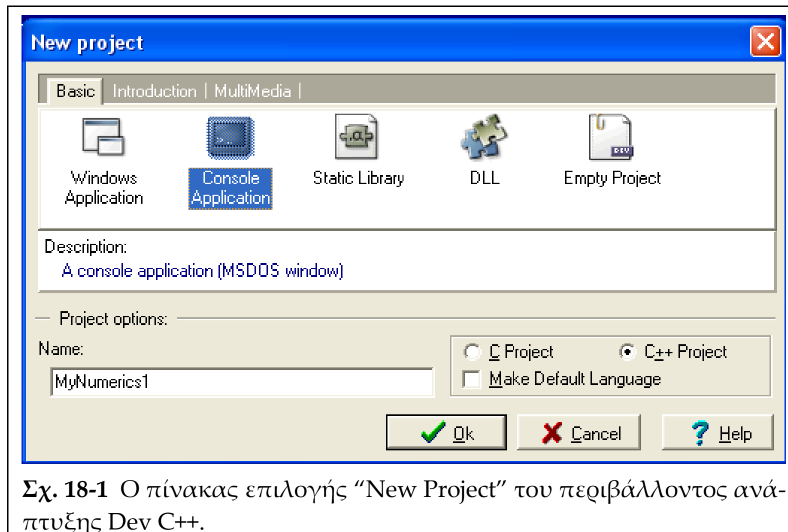
Στο `MyNumerics.cpp` βάζουμε:

```
#include <string>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNumericsXptn( "ph", MyNumericsXptn::noArray );
   if ( m < 0 )
      throw MyNumericsXptn( "ph",
                            MyNumericsXptn::domainError, m );
// τα υπόλοιπα όπως ήταν

void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNumericsXptn( "bisection",
```



Σχ. 18-1 Ο πίνακας επιλογής "New Project" του περιβάλλοντος ανάπτυξης Dev C++.

```
                          MyNumericsXptn::domainError, epsilon );
// τα υπόλοιπα όπως ήταν
```

Παρατηρήσεις: ▶

1. Όπως βλέπεις, τώρα οι συναρτήσεις μας ρίχνουν και εξαιρέσεις. Όταν έχεις να μετατρέψεις μια συνάρτηση που την έχεις αναπτύξει για μια συγκεκριμένη εφαρμογή, όπου η χρήση της είναι –πιθανότατα– ελεγχόμενη, σε συνάρτηση «γενικής χρήσης» θα πρέπει να κάνεις κάτι τέτοιες αλλαγές.
2. Η *bisection()* ρίχνει εξαίρεση αν κληθεί με *epsilon* < 0. Τα υπόλοιπα προβλήματα αντιμετωπίζονται με επιστρεφόμενο κωδικό λάθους.◀

Φυλάγουμε αυτά τα αρχεία στο ευρετήριο `MyNumerics1`.

Στο περιβάλλον ανάπτυξης της Dev-C++ επιλέγουμε:

```
File|New|Project
```

και βλέπουμε τον πίνακα επιλογής που βλέπεις στο Σχ. 18-1. Στο "`Name:`" γράφουμε `MyNumerics1` και επιλέγουμε ως είδος project "`Console Application`". Δίνοντας "`Ok`" μας ζητείται να επιλέξουμε πού θα αποθηκευτεί. Επιλέγουμε το ευρετήριο `MyNumerics1`. Στο `MyNumerics1`, δημιουργείται ένα αρχείο με όνομα `MyNumerics1.dev`. Αυτό είναι το αρχείο του project.

Στο περιβάλλον ανάπτυξης (επιλογή "`Project`" στο αριστερό παράθυρο) βλέπεις ότι έχεις ανοιγμένο το project `MyNumerics1`, που έχει ως περιεχόμενο το αρχείο `main.cpp`· το βλέπεις στο δεξιό παράθυρο:

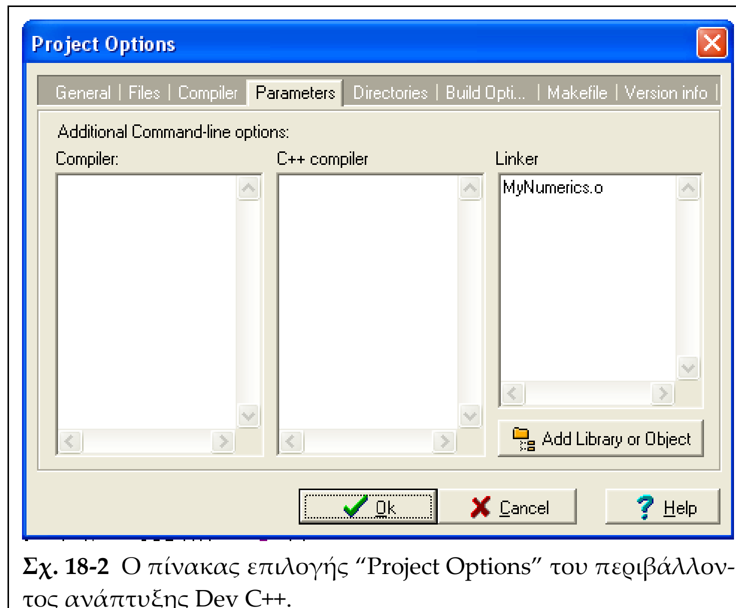
```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Τώρα, επιλέγοντας `Project|Add to Project` ζητούμε να προστεθεί στο project το αρχείο `MyNumerics.cpp`. Βλέπουμε το όνομά του να εμφανίζεται στο αριστερό παράθυρο. Τέλος, επιλέγουμε `Execute|Rebuild all` και βλέπουμε να εμφανίζονται στο `MyNumerics1` τα εξής αρχεία:

```
main.o
Makefile.win
MyNumerics.o
MyNumerics1.exe
```



Σχ. 18-2 Ο πίνακας επιλογής "Project Options" του περιβάλλοντος ανάπτυξης Dev C++.

Μας ενδιαφέρει το `MyNumerics.o` που είναι το «μεταγλωττισμένο `MyNumerics.cpp`».[3] Δες τώρα πώς θα το χρησιμοποιήσουμε.

Απλουστεύουμε το πρόγραμμα της §17.14 τόσο:

```
#include <iostream>
#include <cmath>
#include "MyNumerics.h"

using namespace std;

double q( double x );

int main()
{
   double riza;
   int    errCode;

   bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
   bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
   cout << " Λάθος " << errCode << endl
        << " Ρίζα = " << riza << endl;
}

double q( double x )
{
   return  ( x - log(x) - 2 );
} // q
```

και το φυλάγουμε στο `testBisection.cpp` στο ευρετήριο `testBisection`.

Όπως βλέπεις, το μόνο που υπάρχει από τη *bisection()* είναι η επικεφαλίδα της που υπάρχει στο `MyNumerics.h`. Παρ' όλα αυτά την καλούμε τρεις φορές.

Αντιγράφουμε στο `testBisection` τα `MyNumerics.h` και `MyNumerics.o`.

Και τώρα:

- Δημιουργούμε ένα project είδους `Console Application`, με όνομα `testBisection` και το φυλάγουμε στο ευρετήριο `testBisection`.

---

[3] Αλλού, μπορεί να το δεις ως `MyNumerics.obj`.

- Προσθέτουμε στο project το αρχείο `testBisection.cpp`.
- Κλείνουμε, *χωρίς να φυλάξουμε*, το `main.cpp`. Με τον τρόπο αυτόν το αρχείο διαγράφεται από το project όπου μένει μόνο το `testBisection.cpp`.
- Επιλέγουμε `Project|Project options` και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "`Parameters`".
- Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2) με το "`Add Library or Object`" επιλέγουμε το `MyNumerics.o`. Με αυτόν τον τρόπο ζητούμε από τον συνδέτη, να συνδέσει και το περιεχόμενο αυτού του αρχείου στο εκτελέσιμο που θα δημιουργήσει.
- Για να πάρουμε το εκτελέσιμο, επιλέγουμε `Execute|Rebuild all`.

Στο `testBisection` θα δημιουργηθούν τα εξής αρχεία:

```
testBisection.exe
testBisection.o
```

Το `testBisection.exe` είναι το εκτελέσιμο. Το «`testBisection.o` είναι το μεταγλωττισμένο `testBisection.cpp`».

Στο παράδειγμα αυτό βλέπουμε το εξής: *Το πρόγραμμα αποτελείται από δύο αρχεία που μεταγλωττίζονται ξεχωριστά*. Αν ξαναγυρίσουμε σε αυτά που λέγαμε –για το τι θα δώσουμε στον πελάτη χωρίς να του αποκαλύψουμε τα μυστικά μας– τώρα έχουμε μιαν απάντηση: θα του δώσουμε τα `MyNumerics.h` και `MyNumerics.o`.

Παρατηρήσεις: ▶

1. Χρησιμοποιήσαμε το όνομα "`testBisection`" α) για το project, β) για το ευρετήριο που βάλαμε τα αρχεία και γ) για το αρχείο `cpp` στο οποίο υπάρχει η **main**. Θα μπορούσαμε να χρησιμοποιήσουμε τρία διαφορετικά ονόματα. *Το όνομα του εκτελέσιμου θα είναι ίδιο με αυτό του project.*
2. Στο πρόγραμμά μας εντάσσεται και η *ph* αφού υπάρχει στο `MyNumerics.cpp` επομένως και στο `MyNumerics.o`, παρ' όλο που δεν καλείται από το πρόγραμμά μας. Θα το «ξεφορτωθούμε» στη συνέχεια.
3. Θα μπορούσαμε να είχαμε βάλει ("Add to project") το `MyNumerics.cpp` στο project `testBisection` και να είχαμε και πάλι ξεχωριστή μεταγλώττιση, με την έννοια ότι θα είχαμε δύο ξεχωριστά αρχεία `testBisection.o` και `MyNumerics.o`. Αλλά, σε μια τέτοια περίπτωση μπορεί να σου έμενε η υποψία ότι η **main** «έβλεπε» τη *bisection()*. Τις ξεχωρίσαμε λοιπόν τελείως για να σε πείσουμε.
4. Δες έναν άλλον τρόπο που μπορείς να γράψεις το πρόγραμμά σου:

```
#include <iostream>
#include <cmath>

using namespace std;

extern "C++"
{
   struct MyNumericsXptn;
   void bisection( double (*f)(double),
                   double a, double b, double epsilon,
                   int nMax,
                   double& root, int& errCode );
}

double q( double x );

int main()
// τα υπόλοιπα όπως ήταν
```

Όπως βλέπεις, δεν βάλαμε την `#include "MyNumerics.h"` αλλά βάλαμε την **extern** η οποία δηλώνει ότι σε κάποιο άλλο αρχείο, που θα συνδεθεί στο πρόγραμμα, υπάρχει ένας τύπος δομής με το όνομα *MyNumericsXptn* και μια συνάρτηση με το όνομα *bisection* και την επικεφαλίδα που φαίνεται.

Αν το αρχείο που συνδέουμε έχει βγεί από μεταγλωττιστή C θα έπρεπε να δηλώσουμε "`extern "C"`". Τα διάφορα περιβάλλοντα ανάπτυξης σου επιτρέπουν να συνδέσεις αρχεία που έχουν βγει από μεταγλωττιστές άλλων γλωσσών. Φυσικά, σου επιτρέπουν να κάνεις και την κατάλληλη δήλωση **extern**, π.χ. "`extern "Ada"`", "`extern "FORTRAN"`" κλπ.
5. Δεν είπαμε οτιδήποτε για το `Makefile.win`. Δεν θα πούμε τίποτε περισσότερο από το ότι σε αυτό υπάρχουν οι εντολές για το κτίσιμο του εκτελέσιμου.
6. Αυτά που είπαμε για το περιβάλλον ανάπτυξης Dev C++ ισχύουν περίπου και για το περιβάλλον ανάπτυξης Code::Blocks. Για την Borland C++ v.5.5 σε ένα («μαύρο») παράθυρο `cmd.exe` δώσε:

```
C:\Borland\bc55\bin>bcc32 main.cpp MyNumerics.cpp
```

όπου "`main.cpp`" είναι αυτό που είδαμε πιο πάνω. Έτσι, θα πάρεις το `MyNumerics.obj`. Στη συνέχεια δίνεις:

```
C:\Borland\bc55\bin>bcc32 testBisection.cpp MyNumerics.obj
```

και παίρνεις το "`testBisection.exe`".◀

18.4 Μια Στατική Βιβλιοθήκη

Και τώρα θα κάνουμε μια *στατική βιβλιοθήκη* που θα περιέχει τις δύο συναρτήσεις *ph()* και *bisection()*.

Στο ευρετήριο `MyNumerics` αντιγράφουμε και πάλι τα `MyNumerics.cpp` και `MyNumerics.h` και –όπως περιγράψαμε στην προηγούμενη παράγραφο– δημιουργούμε ένα project με το όνομα `MyNumerics` αλλά αυτή τη φορά διαφορετικού είδους: "`Static Library`". Προσθέτουμε (Add to project) το `MyNumerics.cpp` και δίνουμε `Execute|Rebuild all`. Στο `MyNumerics`, εκτός από τα `MyNumerics.dev` και `MyNumerics.o`, θα δεις και το `MyNumerics.a`. Αυτή είναι η στατική βιβλιοθήκη[4] που θα χρησιμοποιήσουμε για να ξαναχτισουμε το πρόγραμμά μας.

Στο ευρετήριο `testBisectionL` αντιγράφουμε τα

```
testBisection.cpp
MyNumerics.a
MyNumerics.h
```

Στη συνέχεια:

- Δημιουργούμε ένα project "`Console Application`"με το όνομα `testBisectionL` και
- προσθέτουμε το `testBisection.cpp` (κλείνουμε χωρίς να φυλάξουμε το `main.cpp`).
- Επιλέγουμε `Project|Project options` και από τον πίνακα που εμφανίζεται κάνουμε την επιλογή "`Parameters`". Στο τρίτο από τα «κουτιά» που βλέπουμε (Σχ. 18-2), με το "`Add Library or Object`", επιλέγουμε τη βιβλιοθήκη `MyNumerics.a`.
- Τέλος, επιλέγουμε `Execute|Rebuild all` για να πάρουμε το εκτελέσιμο που θα έχει το όνομα `testBisectionL.exe`,

Αυτό το εκτελέσιμο διαφέρει από το προηγούμενο ως προς το ότι δεν έχει και τη συνάρτηση *ph()*, αλλά μόνον τη *bisection()*.

Όπως καταλαβαίνεις, στον πελάτη, για τον οποίον συζητούσαμε, θα δώσουμε το `MyNumerics.a` και το `MyNumerics.h`.

---

[4] Μια άλλη συνηθισμένη κατάληξη (π.χ. από τη Borland C++) για στατικές βιβλιοθήκες είναι η "`.lib`".

18.5 “namespace”: Το Πρόβλημα και η Λύση

Ας πούμε τώρα ότι γράψαμε τη βιβλιοθήκη περιγραμμάτων, γράψαμε τη στατική βιβλιοθήκη μας και θέλουμε να τις χρησιμοποιήσουμε σε ένα πρόγραμμα που γράφουμε μαζί με δυο βιβλιοθήκες που «κατεβάσαμε» από το Internet. Και ξαφνικά, ο μεταγλωττιστής διαμαρτύρεται: βρίσκει δύο συναρτήσεις με το όνομα *renew()* και δύο συναρτήσεις με το όνομα *ph()*. Και τώρα τι γίνεται; Καλά, άντε και αλλάζουμε αυτά τα δύο ονόματα. Είναι σχεδόν σίγουρο ότι μετά από λίγο θα έχουμε άλλο παρόμοιο πρόβλημα. Μηπως να βάλουμε κάποια «εξωφρενικά» ονόματα στις συναρτήσεις μας. Ε, όχι και έτσι...

Η C++ προσφέρει μια λύση για το πρόβλημα: τον **ονοματοχώρο** (namespace). Δες ένα παράδειγμα. Σε κάποιο πρόγραμμα έχουμε:

```
namespace test
{
   const double x = 1.5;
}
char x = 'A';
int main()
{
   int x = 4;
      :
   cout << x << "  " << ::x << "  " << test::x << endl;
```

Εδώ βλέπουμε:

- Έναν ονοματοχώρο, με όνομα *test*, όπου δηλώνεται μια σταθερά τύπου `double` με όνομα *x* και τιμή 1.5.
- Μια δήλωση της καθολικής μεταβλητής *x* τύπου `char` με αρχική τιμή `'A'`.
- Μια δήλωση μεταβλητής τύπου `int`, τοπικής στην `main`, με *x* όνομα αρχική τιμή 4.

Από τη `main` μπορούμε να έχουμε πρόσβαση και στα τρία παραπάνω αντικείμενα, όπως φαίνεται και από την εντολή εξόδου που δίνει:

```
4  A  1.5
```

Όπως ήδη ξέρουμε, με τα `x` και `::x` παίρνουμε την τοπική και την καθολική μεταβλητή αντιστοίχως. Με το `test::x` εννοούμε: το αντικείμενο με το όνομα *x* που δηλώνεται στον ονοματοχώρο *test*. Όπως βλέπεις, ο τελεστής “`::`” είναι ο τελεστής **επίλυσης** των προβλημάτων **εμβέλειας** (scope resolution) στο πρόγραμμά μας

Έστω λοιπόν ότι μας δίνεται μια βιβλιοθήκη προγραμμάτων. Αν ξέρεις ότι μέσα στη βιβλιοθήκη υπάρχει δηλωμένος κάποιος ονοματοχώρος δεν έχεις πρόβλημα. Αν δεν υπάρχει ονοματοχώρος –αν π.χ. είναι προγράμματα C– μπορείς να κάνεις το εξής: αν, ας πούμε, οι δηλώσεις της βιβλιοθήκης υπάρχουν στο αρχείο `bib.h` δηλώνεις:

```
namespace xbib
{
#include "bib.h"
}
```

Ύστερα από αυτό μέσα στο πρόγραμμά σου χρησιμοποιείς οτιδήποτε δηλώνεται στο bib.h με το πρόθεμα “`xbib::` ”.

Φυσικά, η λύση που δίνεται με το **namespace** μπορεί να δημιουργήσει το αντίθετο πρόβλημα: να θέλεις να χρησιμοποιήσεις μια βιβλιοθήκη, να μην υπάρχει πρόβλημα με τα ονόματα και παρ' όλα αυτά να έχεις την υποχρέωση να γράφεις το όνομα του ονοματοχώρου πριν από όλα τα αντικείμενα που χρησιμοποιείς. Υπάρχει λύση και γι' αυτό. Ας πούμε ότι χρησιμοποιείς μια βιβλιοθήκη, που οι δηλώσεις της υπάρχουν στο `prlib.h` στον ονοματοχώρο *prlib*. Αν βάλεις στο πρόγραμμά σου:

```
#include "prlib.h"
using namespace prlib;
```

μπορείς να χρησιμοποιείς οτιδήποτε υπάρχει στον ονοματοχώρο *prlib* χωρίς το πρόθεμα “`prlib::`”.

Αντί για την καθολική οδηγία "**using namespace**" που κάνει ορατά τα πάντα παντού, υπάρχει και η δήλωση **using** με την οποία μπορείς να κάνεις πιο επιλεκτική δουλειά. Ας πούμε ότι εσύ θέλεις στη συνάρτηση *f1()* να χρησιμοποιήσεις τον τύπο *cl1* της *prlib* ενώ στη συνάρτηση *f2()* θέλεις να χρησιμοποιήσεις τη συνάρτηση *fl0()* της *prlib*. Μπορείς να γράψεις:

```
   double f1(...)
   {
      using prlib::cl1;
      cl1 a, b;
          :
   } // f1

   void f2(...)
   {
      using prlib::fl0;
          :
      q = fl0(1, u) + w;
   } // f2
```

Η δήλωση "`using prlib::cl1`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα `cl1` μέσα στη συνάρτηση *f1()* μόνον. Παρομοίως, η δήλωση "`using prlib::fl0`" μας δίνει τη δυνατότητα να χρησιμοποιούμε χωρίς πρόθεμα το όνομα "`fl0`" μέσα στη συνάρτηση *f2()* μόνον. Για τη δήλωση **using** ισχύουν οι κανόνες εμβέλειας που ξέρουμε.

Τώρα μπορείς να καταλάβεις και τις δηλώσεις "`using namespace std`" που βάζουμε στα προγράμματά μας. Η C++ έχει τον δικό της πάγιο ονοματοχώρο με το όνομα "`std`" (**st**andar**d**). Μέσα σε αυτόν είναι δηλωμένα τα πάντα. Αν δεν το βάζαμε θα έπρεπε να γράφουμε `std::strncpy`, `std::cout`, `std::endl`, `std::cin`, `std::bad_alloc`, κλπ.

Ας έλθουμε τώρα στα παραδείγματά μας. Στο αρχείο `MyTmpltLib.h` κάνουμε την εξής αλλαγή:

```
namespace MyTmplt
{
struct MyTmpltLibXptn
// . . .
template< typename T >
int linSearch( const T v[], int n,
               int from, int upto, const T& x )
// . . .
template< typename T >
void renew( T*& p, int ni, int nf )
// . . .
template< typename T >
T** new2d( int nR, int nC )
// . . .
template< typename T >
void delete2d( T**& a, int nR )
// . . .
} // namespace MyTmplt
```

Έτσι, τα ονόματα: *MyTmpltLibXptn*, *linSearch*, *renew*, *new2d* και *delete2d* εισάγονται στον ονοματοχώρο *MyTmplt*.

Εδώ όμως πρόσεξε το εξής:

- ***Ο ορισμός δομής (κλάσης) ορίζει και έναν ονοματοχώρο με το όνομα της δομής.***

Έχουμε λοιπον τον ονοματοχώρο *MyTmpltLibXptn* φωλιασμένο μέσα στον ονοματοχώρο *MyTmplt*. Έτσι, στη διαχείριση εξαιρέσεων στο πρόγραμμα του Παράδ. 2 της §16.13 οι κωδικοί σφάλματος θα έχουν «διπλό πρόθεμα», για παράδειγμα,

"`MyTmplt::MyTmpltLibXptn::domainError`"

```
// . . .
   catch( MyTmplt::MyTmpltLibXptn& x )
   {
      switch ( x.errCode )
      {
```

```
         case MyTmplt::MyTmpltLibXptn::domainError:
            cout << x.funcName << "called with parameters "
                 << x.errVal1 << ", " << x.errVal2 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::noArray:
            cout << x.funcName << "called with NULL pointer"
                 << endl;
            break;
         case MyTmplt::MyTmpltLibXptn::allocFailed:
            cout << "cannot get enough memory " << " in "
                 << x.funcName << endl;
            break;
         default:
            cout << "unexpected MyTmpltLibXptn from "
                 << x.funcName << endl;
      } // switch
   } // catch( MyTmpltLibXptn
// . . .
```

Ακόμη, στην *elmntInTable* θα έχουμε:

```
// . . .
   int lPos( MyTmplt::linSearch( grElmnTbl, nElmn,
                                 0, nElmn-1, oneElmn) );
   if ( lPos < 0 )
   {
      if ( nElmn+1 == nReserved )
      {
         MyTmplt::renew( grElmnTbl, nElmn, nReserved+incr );
// . . .
```

Ας δούμε τώρα πώς θα δηλώσουμε ονοματοχώρο στη στατική βιβλιοθήκη μας. Στο αρχείο `MyNumerics.h` κάνουμε την εξής αλλαγή:

```
namespace MyNmr
{
struct MyNumericsXptn
// . . .
double ph( const double a[], int m, double x );
// . . .
void bisection( double (*f)(double),
                double a, double b, double epsilon,
                int nMax,
                double& root, int& errCode );
} // namespace MyNmr
```

ενώ στο `MyNumerics.cpp` θα πρέπει να βάλουμε:

```
// . . .
double MyNmr::ph( const double a[], int m, double x )
{
   if ( a == 0 )
      throw MyNmr::MyNumericsXptn( "ph",
                                 MyNmr::MyNumericsXptn::noArray );
// . . .
void MyNmr::bisection( double (*f)(double),
                       double a, double b, double epsilon,
                       int nMax,
                       double& root, int& errCode )
{
   if ( epsilon < 0 )
      throw MyNmr::MyNumericsXptn( "linSearch",
                     MyNmr::MyNumericsXptn::domainError, epsilon );
// . . .
```

Δεν θα μπορούσαμε να βάλουμε στο `MyNumerics.cpp` μια

```
using namespace MyNmr;
```

και να γλυτώσουμε από όλα αυτά τα "`MyNmr`"; Όχι! Οι ορισμοί των δύο συναρτήσεων είναι πλήρεις και ο μεταγλωττιστής θα τις μεταγλωττίσει ως *ph()* και *bisection()* και όχι ως *MyNmr* ::*ph()* και *MyNmr*::*bisection()*.

Στο `testBisection.cpp` θα πρέπει να βάλουμε:

```
MyNmr::bisection( q, 0.1, 1.0, 1e-20, 1000000, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 5, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
MyNmr::bisection( q, 0.1, 1.0, 1e-10, 50, riza, errCode );
cout << " Λάθος " << errCode << endl
     << " Ρίζα = " << riza << endl;
```

Αν είχαμε και διαχείριση εξαιρέσεων θα βάζαμε:

```
catch( MyNmr::MyNumericsXptn& x )
{
   if ( x.errCode == MyNmr::MyNumericsXptn::domainError )
      cout << x.funcName << " called with negative ( "
           << x.errDblVal << " ) epsilon" << endl;
   else
      cout << "unexpected MyNumericsXptn from "
           << x.funcName << endl;
}
```

Αν θέλεις μπορείς να χρησιμοποιήσεις τον ίδιο ονοματοχώρο για πολλές βιβλιοθήκες. Για παράδειγμα, αντί για τα δύο ονόματα, *MyTmplt* και *MyNmr*, θα μπορούσαμε να χρησιμοποιήσουμε ένα μόνο όνομα, ας πούμε *MyNmspc*.

Γενικώς, να θυμάσαι ότι:

♦ ***Μια καλή βιβλιοθήκη έχει και τον ονοματοχώρο της.***

Αυτό μπορεί να σε γλυτώσει από προβλήματα χωρίς να σου δημιουργεί νέα αφού αρκεί μια "`using namespace …`" για να σου επιτρέψει να τον αγνοήσεις όπου δεν τον χρειάζεσαι.

18.6 Ανακεφαλαίωση

Οι βιβλιοθήκες συναρτήσεων είναι πολύτιμα εργαλεία για τον προγραμματιστή. Στο διαδίκτυο υπάρχουν πολλές και αρκετές από αυτές είναι πολύ καλής ποιότητας. Μπορείς να κτίσεις και δικές σου.

Οι *βιβλιοθήκες περιγραμμάτων συναρτήσεων* είναι οι πιο απλές αφού το περιεχόμενό τους είναι γραμμένο σε C++. Περιέχουν περιγράμματα συναρτήσεων, εξειδικεύσεις τους και πιθανότατα κάποια κλάση εξαιρέσεων. Καλό είναι να δηλώνονται μέσα σε έναν *ονοματοχώρο*.

Πριν δούμε τις βιβλιοθήκες μεταγλωττισμένων συναρτήσεων είδαμε πώς μπορούμε να μεταγλωττίζουμε χωριστά διάφορα κομμάτια του προγράμματος και με τη σύνδεση να παίρνουμε το εκτελέσιμο.

Οι βιβλιοθήκες μεταγλωττισμένων συναρτήσεων χωρίζονται σε δύο μεγάλες κατηγορίες: τις *στατικές* και τις *δυναμικής σύνδεσης*.

- Στατικές: Οι συναρτήσεις τους, που καλούνται από κάποιο πρόγραμμα, ενσωματώνονται (αντιγράφονται) στο εκτελέσιμο που μπορεί να εκτελείται χωρίς να είναι παρούσα η βιβλιοθήκη.
- Δυναμικής Σύνδεσης: Οι συναρτήσεις τους δεν ενσωματώνονται στο εκτελέσιμο αλλά φορτώνονται στη μνήμη όταν αυτό εκτελείται.

Για τη χωριστή μεταγλώτιση και τη δημιουργία και τη χρήση βιβλιοθηκών τηρούνται διαδικασίες που εξαρτώνται σε μεγαλύτερο ή μικρότερο βαθμό από το περιβάλλον ανάπτυξης ή/και το ΛΣ.

Όταν έχεις χωριστή μεταγλώττιση ή χρησιμοποιείς βιβλιοθήκες συναρτήσεων είναι απαραίτητα τα αρχεία επικεφαλίδων (`.h`).

Ασκήσεις

Α Ομάδα

18-1 Στις συναρτήσεις *ph()* και *bisection()* αντιστοιχίσαμε στο ℝ τον **`double`**. Θα μπορούσαμε να είχαμε αντιστοιχίσει τον **`float`** ή τον **`long double`**. Αν κάνουμε τις συναρτήσεις περιγράμματα δίνουμε στον προγραμματιστή τη δυνατότητα να επιλέξει τον τύπο για την περίπτωση που τον ενδιαφέρει.

Αφού μετατρέψεις τις δύο συναρτήσεις σε περιγράμματα, να τις εντάξεις σε μια βιβλιοθήκη περιγραμμάτων `NumTmplt`. Γράψε προγράμματα για να τη δοκιμάσεις.

Β Ομάδα

18-2 Στο Κεφ. 15 είδαμε τον τύπο

```
struct GrElmn
{
   enum { symbolSz = 4, nameSz = 14, grNameSz = 14,
          saveSize = sizeof(short int) + sizeof(float) +
                     symbolSz + nameSz + grNameSz };
   unsigned short int geANumber;     // ατομικός αριθμός
   float              geAWeight;     // ατομικό βάρος
   char               geSymbol[symbolSz];
   char               geName[nameSz];
   char               geGrName[grNameSz];
}; // GrElmn
```

και τις συναρτήσεις

```
GrElmn GrElmn_copyFromElmn( const Elmn& a );
void GrElmn_save( const GrElmn& a, ostream& bout );
void GrElmn_load( GrElmn& a, istream& bin );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_display( const GrElmn& a, ostream& tout );
void GrElmn_setGrName( GrElmn& a, string newGrName );
void GrElmn_writeToTable( const GrElmn& a, ostream& tout );
```

για τη διαχείριση των στοιχείων του.

Κάνε μια στατική βιβλιοθήκη `GrElmnLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις. Δοκίμασέ την στα προγράμματα των §15.14.1 και 15.14.2.

Προσοχή! Δυο συναρτήσεις ρίχνουν εξαιρέσεις *ApplicXptn*. Αυτό πρέπει να αλλάξει: θα ρίχνουν εξαιρέσεις `GrElmnXptn`. Φυσικά, αυτόν τον τύπο θα τον ορίσεις εσύ!

18-3 Στο Κεφ. 15 είδαμε και τον τύπο

```
struct Date
{
   enum { saveSize = sizeof(int) + 2*sizeof(char) };
   unsigned int  year;
   unsigned char month;
   unsigned char day;
   Date( int yp=1, int mp=1, int dp=1 )
   {  year = yp;  month = mp;  day = dp;  }
}; // Date
```

του οποίου τα στοιχεία διαχειριστήκαμε με τις συναρτήσεις

```
void Date_save( const Date& a, ostream& bout );
```

```
void Date_load( Date& a, istream& bin );
```

και τους τελεστές

```
bool operator==( const Date& lhs, const Date& rhs );
bool operator<( const Date& lhs, const Date& rhs );
ostream& operator<<( ostream& tout, const Date& rhs );
```

Κάνε μια στατική βιβλιοθήκη `DateLib` (`.a` ή `.lib`) με αυτές τις συναρτήσεις και τους τελεστές. Δοκίμασέ την σε αυτά που λέγαμε για τον τύπο *Date*.

Ξαναλύσε την Άσκ. 15-1 με χρήση της βιβλιοθήκης.

Προσοχή! Οι συναρτήσεις αυτές δεν ρίχνουν εξαιρέσεις αλλά θα έπρεπε. Δες την προηγούμενη άσκηση και τις (αντίστοιχες) συναρτήσεις της.