ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Τεχνικές Προγραμματισμού με βάση την C

Γιώργος Β. Μπρούσκαρης
Βασίλειος Α. Καρανδρέας

Επιβλέπων: **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2017

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Τεχνικές Προγραμματισμού με βάση την C

Γεώργιος Β. Μπρούσκαρης
Α.Μ.: 1115199901265
Βασίλειος Α. Καρανδρέας
Α.Μ.: 1115200200059

ΕΠΙΒΛΕΠΩΝ: **Ιωάννης Κοτρώνης,** Αναπληρωτής Καθηγητής

ΠΕΡΙΛΗΨΗ

Το αντικείμενο της εργασίας είναι η ανάλυση διαφόρων τεχνικών προγραμματισμού, οι οποίες αν ακολουθηθούν θα έχουν σαν αποτέλεσμα να γράφεται σωστός και κατανοητός κώδικας, τόσο από εμάς όσο και από τρίτους που δεν το έχουν ξαναδεί.

Για να επιτευχθεί αυτό αναλύουμε στην αρχή την μορφή την οποία θα πρέπει να έχει ο κώδικάς μας, καθώς και την συμπεριφορά που πρέπει να έχει το πρόγραμμα μας ώστε να γίνεται κατανοητό. Συνεχίζουμε παρουσιάζοντας τι είναι δομημένος προγραμματισμός και αναλύοντας μεθόδους σχεδιασμού των προγραμμάτων καθώς και το πώς να ελέγχουμε τα μη σωστά προγράμματα. Τέλος αναλύουμε το πως πρέπει να είναι ένα καλοσχεδιασμένο δομικό στοιχείο (module) σε ένα πρόγραμμα.

Τα κύρια αποτελέσματα της εργασίας μας είναι ότι είναι ωφέλιμο να δαπανηθεί κάποια ώρα, πριν ακόμα ξεκινήσουμε να γράφουμε τον κώδικα μας, ώστε να γίνει σωστή ανάλυση και σχεδιασμός του προγράμματος μας γιατί με αυτόν τον τρόπο θα γλιτώσουμε πολλαπλάσιο χρόνο στην συνέχεια. Επίσης καταλήξαμε ότι αν ακολουθηθούν κάποιοι απλοί κανόνες για το γράψιμο του κώδικα τότε ο κώδικας γίνεται πολύ πιο ευανάγνωστος και θα αποφευχθούν πολλά λάθη και χρόνος. Τέλος είδαμε ότι αν αποφασίσουμε να διορθώσουμε όλα τα λάθη του κώδικα και της λογικής μας στο τέλος, αυτή θα είναι μια εξαιρετικά κουραστική, επίπονη και χρονοβόρα διαδικασία.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνικές προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Μορφή προγραμματισμού, top-down, Testing, Modularity, Generics

ABSTRACT

The subject of the thesis is the analysis of various programming techniques, which if followed will result in written code that will be equally correct and comprehensible to us and any third parties who have not seen it before.

In order to achieve this, we first analyze the format that our source code should have, as well as the behavior of the whole program so that it can be apprehended. We proceed to explain what is structured programming and we follow up by analyzing program design methods and how to fix faulty programs. Finally, we analyze how a well-designed module should be represented in our program.

The main conclusion of our thesis is that it is beneficial to spend some time before starting to write the source code so that you can properly analyze and design your program. This has been proven to save a lot of time in the long term. We have also come to the conclusion that if some simple code rules are followed, then the code becomes much easier to read, and many errors and time will be avoided. Finally we have observed that if we decide to correct all the errors of our code and fix the logic issues in the end, it will be an extremely tedious, laborious and time-consuming process.

SUBJECT AREA: Programming Techniques

KEYWORDS: Programming format, top-down, Testing, Modularity, Generics

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα θέλαμε να ευχαριστήσουμε τον επιβλέποντα αναπληρωτή καθηγητή Ιωάννη Κοτρώνη, για τη συνεργασία, την υπομονή του, τον χρόνο του και την πολύτιμη συμβολή του στην ολοκλήρωση της.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΠΡΟΛΟΓΟΣ

Η εργασία διενεργήθηκε με σκοπό να βοηθηθούν οι πρωτοετείς φοιτητές του τμήματος μας στην αρχή του δεύτερου εξαμήνου ώστε να μπορέσουν να ανταποκριθούν στις απαιτήσεις του μαθήματος δομές δεδομένων αλλά και να μάθουν μερικές καλές πρακτικές που θα τους βοηθήσουν σε κάθε μελλοντικό τους προγραμματιστικό εγχείρημα.

Η διεξαγωγή της εργασίας έγινε στην Αθήνα από τους φοιτητές Μπρούσκαρη Γιώργο και Καρανδρέα Βασίλη με την βοήθεια του αναπληρωτή καθηγητή Ιωάννη Κοτρώνη.

1. ΕΙΣΑΓΩΓΗ

Το αντικείμενο της εργασίας είναι η ανάλυση διαφόρων τεχνικών προγραμματισμού, οι οποίες αν ακολουθηθούν θα έχουν σαν αποτέλεσμα να γράφεται σωστός και κατανοητός κώδικας, τόσο από εμάς όσο και από τρίτους που δεν το έχουν ξαναδεί.

Για να επιτευχθεί αυτό αναλύουμε στην αρχή την μορφή την οποία θα πρέπει να έχει ο κώδικάς μας, καθώς και την συμπεριφορά που πρέπει να έχει το πρόγραμμα μας ώστε να γίνεται κατανοητό. Συνεχίζουμε παρουσιάζοντας τι είναι δομημένος προγραμματισμός και αναλύοντας μεθόδους σχεδιασμού των προγραμμάτων καθώς και το πώς να ελέγχουμε τα μη σωστά προγράμματα. Τέλος αναλύουμε το πως πρέπει να είναι ένα καλοσχεδιασμένο δομικό στοιχείο (module) σε ένα πρόγραμμα.

Πιο αναλυτικά στο δεύτερο κεφάλαιο ασχολούμαστε με την μορφή του προγραμματισμού, όπως τους κανόνες που πρέπει να διέπουν την ονοματολογία, τα σχόλια και την χρήση των εμφωλευμένων δομών.

Στο τρίτο κεφάλαιο αναλύουμε την συμπεριφορά των προγραμμάτων τόσο κατά την εκτέλεση τους όσο και στο τελείωμα τους, καθώς και διάφορες προχωρημένες τεχνικές.

Στο τέταρτο κεφάλαιο μελετάμε τον δομημένο προγραμματισμό και τις ακολουθιακές, υποθετικές και επαναληπτικές εντολές.

Στο πέμπτο κεφάλαιο αναλύουμε τις βασικές αρχές του testing και τις υποκατηγορίες του(internal, external) καθώς και γενικές στρατηγικές του. Τέλος βλέπουμε βασικά πράγματα για το debugging.

Στο έκτο κεφάλαιο ασχολούμαστε με μεθόδους σχεδιασμού προγραμμάτων. Αναλύουμε τον top-down προγραμματισμό και βλέπουμε τα πλεονεκτήματα αυτής της μεθόδου.

Στο έβδομο κεφάλαιο ασχολούμαστε με το modularity και πιο συγκεκριμένα με τα χαρακτηριστικά του, τα οποία τα αναλύουμε.

Τέλος στο όγδοο κεφάλαιο ασχολούμαστε με την γενικότητα(generics) όπου βλέπουμε παραδείγματα για διάφορους τύπους δεδομένων και παραδείγματα για γενικούς αλγόριθμους.

2. ΜΟΡΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Λέγοντας μορφή προγραμματισμού (programming style) εννοούμε το σύνολο των συμβάσεων, κανόνων, βοηθημάτων και οδηγιών που κάνουν ένα πρόγραμμα ευκολότερο στην ανάγνωση και την κατανόηση για τους ανθρώπους. Ένα γεγονός που συχνά παραβλέπετε είναι ότι τα υπολογιστικά προγράμματα χρησιμοποιούνται τόσο από τους ανθρώπους όσο και από τις μηχανές.

Όλοι αυτοί οι κανόνες μορφοποίησης ενός προγράμματος είναι περιττοί για μία μηχανή αφού αυτή ενδιαφέρεται μόνο για την τήρηση των συντακτικών κανόνων και όχι για την καλή αναγνωσιμότητα ενός προγράμματος. Σαν ένα παράδειγμα βλέπουμε δύο κομμάτια ενός προγράμματος που κάνουν την ίδια δουλειά. Τα προγράμματα από την μεριά της μηχανής είναι ισοδύναμα αφού καταλήγουν στο ίδιο αποτέλεσμα και χρησιμοποιούν σωστούς συντακτικούς κανόνες. Από την πλευρά του ανθρώπου όμως το δεύτερο είναι πιο δυσνόητο και επομένως διαβάζεται και αναπτύσσεται πιο δύσκολα.

```
if( 2 * k < n - m )
   *xp = c[k+1];
else
   *xp = d[k--];
*x += *xp;
```

a)

```
*x += ( *xp = (2 * k ) < ( n - m) ? c[k+1]  : d[k--]); [1]
```

b)

Η αναγκαιότητα της τήρησης κανόνων στην μορφή των προγραμμάτων είναι ιδιαίτερα έντονη σε ένα περιβάλλον παραγωγής παρά σε ένα ακαδημαϊκό περιβάλλον. Αυτό γίνεται φανερό από τις διαφορές που υπάρχουν ανάμεσα σε ένα ακαδημαϊκό περιβάλλον και σε ένα περιβάλλον παραγωγής όσον αφορά την δομή, την εκτέλεση και την ανάπτυξη των προγραμμάτων.

Πίνακας 1: Διαφορές ακαδημαϊκού περιβάλλοντος και περιβάλλοντος παραγωγής

ΑΚΑΔΗΜΑΪΚΟ	ΠΑΡΑΓΩΓΗ
Τα προγράμματα εκτελούνται συνήθως μια φορά και σχεδόν ποτέ δεν τροποποιούνται μετά από την τελική εκτέλεση.	Τα ίδια προγράμματα εκτελούνται για μακρά χρονικά διαστήματα μερικές φορές περισσότερο και από τον χρόνο απασχόλησης των προγραμματιστών που τα υλοποιήσαν αρχικά.
Τα προγράμματα αυτά εξετάζονται συνήθως σε λεπτομέρεια από λίγα άτομα και συνήθως μόνο από τον φοιτητή και τον επιβλέποντα καθηγητή.	Τα προγράμματα διαβάζονται, ελέγχονται και αναπτύσσονται από πολλά άτομα αφού το παραγωγικό δυναμικό συχνά αλλάζει.
Τα προγράμματα δεν αλλάζουν. Συντηρούνται στην ίδια μορφή μόλις τρέξουν μια φορά σωστά.	Τα προγράμματα αλλάζουν αρκετά συχνά αφού αλλάζουν συχνά στοιχεία που συσχετίζονται με αυτά όπως οι φορολογικοί νόμοι, η σχετική νομοθεσία, οι λογιστικές στρατηγικές, οι τρόποι παραγωγής, οι καταναλωτικές προτιμήσεις και άλλα στοιχεία.
Δεν υπάρχει οικονομικός προϋπολογισμός για τα προγράμματα και ο φοιτητής δουλεύει χωρίς χρονικούς περιορισμούς μέχρι να τελειώσει.	Υπάρχουν αυστηρά χρονικά περιθώρια και όρια προϋπολογισμού.

Τα προγράμματα είναι μικρά και υλοποιούνται συνήθως από 1 άτομο.	Μεγάλα προγράμματα συνήθως πολλών χιλιάδων γραμμών που υλοποιούνται από αρκετά άτομα 3-10 ή από ομάδες εργασίας.

Όλες οι μέθοδοι μορφοποίησης των προγραμμάτων έχουν ως σκοπό**:**

- Την αποφυγή χαμένου χρόνου και χρήματος για τον εντοπισμό και την διόρθωση λαθών.
- Την δημιουργία ευανάγνωστων και κατανοητών προγραμμάτων.
- Την ευκολία συντήρησης και επέκτασης ενός προγράμματος.
- Την εύκολη συνεργασία μεταξύ των προγραμματιστών.

Η συζήτηση γύρω από την μορφή των προγραμμάτων αφορά μόνο οδηγίες και συστάσεις και όχι κανόνες. Στα επόμενα υποκεφάλαια θα δούμε το τι πρέπει να αποφεύγουμε και θα δίνονται οδηγίες πώς να γράφονται εναλλακτικά μέσω του προγράμματος που υπάρχει στο τέλος του κεφαλαίου.

2.1 Διαύγεια και απλότητα έκφρασης

Οι προγραμματιστές θα πρέπει γενικά να μάθουν να μειώνουν το μέγεθος των επιθυμιών τους προς όφελος της διαύγειας και της απλότητας ενός προγράμματος το οποίο θα χρησιμοποιηθεί και από άλλους χρήστες. Ειδικότερα θα πρέπει να μην θυσιάζουν την διαύγεια έκφρασης για «έξυπνο» προγραμματισμό και να μην θυσιάζουν την διαύγεια έκφρασης για μικρές μειώσεις στον χρόνο εκτέλεσης ενός προγράμματος. Ακόμα να αποφεύγουν προγραμματιστικά κόλπα όπως:

num – (num / i) * i αντί να γράφουν num % i

και να προσπαθούν να γράφουν προγράμματα που να διακρίνονται για την απλότητα τους.

2.2 Ονοματολογία

Κύρια συνεισφορά στην καλή αναγνωσιμότητα και στην σαφήνεια ενός προγράμματος έχει η χρησιμοποίηση μνημονικών ονομάτων στις μεταβλητές (variables). Ειδικότερα σε ότι αφορά την ονοματολογία οι προγραμματιστές θα πρέπει να:

- επιλέγουν καλά περιγραφικά ονόματα για τις συναρτήσεις(functions) και τις μεταβλητές.

Παράδειγμα να γράφουν:

```
final_price = initial_price + initial_price * (tax / 100);
```

αντί για:

```
fp = ip + ip * (t / 100);
```

τα οποία δεν διευκρινίζουν τον σκοπό των πράξεων και τον ρόλο των μεταβλητών που χρησιμοποιούν.

- χρησιμοποιούν ονόματα αρκετά μεγάλα που να ξεκινούν με κεφαλαίο γράμμα για τις καθολικές μεταβλητές, ώστε να θυμίζουν στον χρήστη την σημασία τους καθώς μπορούν να χρησιμοποιηθούν οπουδήποτε στον κώδικα, όπως: [2]

```
int Sum_of_divisors ;
```

- χρησιμοποιούν τυποποιημένα προθέματα και καταλήξεις για τις αντίστοιχες μεταβλητές όπως weeklygross, yeargross, masterfile, transactionfile, updatefile.
- δίνουν ονόματα σε σταθερές οι οποίες να γράφονται με κεφαλαία γράμματα [3].

Για παράδειγμα αντί για:

```
for( currentnum = 2; currentnum < 400000; currentnum++)
```

να γράφουν

```
#define MAXNUM 400000
for( currentnum = 2; currentnum < MAXNUM; currentnum++)
```

- χρησιμοποιούν ονόματα που να αρχίσουν ή να τελειώνουν με p, όπως nodep ή pnode για τους δείκτες (pointers).
- δίνουν ενεργά ονόματα στις συναρτήσεις. Τα ονόματα των συναρτήσεων πρέπει να είναι ενεργά ρήματα, που ίσως να ακολουθούνται από ουσιαστικά [4]. Παράδειγμα:

```
int calculate_sum_of_divisors(int num);
```

Σε περίπτωση που το όνομα της μεταβλητής είναι μεγάλο να χρησιμοποιηθεί η υπογράμμιση ¨_¨. Παράδειγμα, αντί η μεταβλητή μας να έχει το όνομα Sumofdivisors να έχει το όνομα Sum_of_divisors.

Επίσης θα πρέπει να αποφεύγουν να:

- χρησιμοποιούν κρυπτογραφημένα και ασαφή ονόματα και συντομεύσεις που δεν είναι κοινά αποδεκτά. Όπως: sod(Sum_of_divisors).
- χρησιμοποιούν μικρά ονόματα προς χάριν λιγότερης πληκτρολόγησης.
- χρησιμοποιούν 'χαριτωμένα' ονόματα που δεν έχουν μνημονικό χαρακτήρα, όπως :

```
if (puss_in_boots){
   ...
}
```

- χρησιμοποιούν ονόματα μεταβλητών που μπορεί εύκολα να μπερδευτούν. Ειδικά να είστε προσεχτικοί με την ταυτόχρονη χρήση σε διαφορετικές μεταβλητές των 0 και o, 2 και z, 1 και i. Επίσης μερικές γραμματοσειρές έχουν ίδια εμφάνιση για το 1 και το l. Για παράδειγμα οι μεταβλητές:

```
int xxxyz;
```

και

```
int xxxy2;
```

μπορεί εύκολα να μπερδευτούν.

2.3 Σχόλια (Comments)

Στο ακαδημαϊκό περιβάλλον τα σχόλια στα προγράμματα είναι μεν επιθυμητά αλλά όχι αναγκαία εξαιτίας των μικρών προγραμμάτων και των λίγων ανθρώπων που απασχολεί η ολοκλήρωση τους. Σε αντίθεση σε ένα περιβάλλον παραγωγής, που η καλή αναγνωσιμότητα και η συντήρηση των προγραμμάτων είναι καθοριστικά στοιχεία, είναι σημαντικό να περιλαμβάνουμε σχολιασμούς των προγραμμάτων είτε αυτοί αφορούν ολόκληρο το πρόγραμμα είτε ενότητες τους.

Ιδιαίτερη μνεία θα πρέπει να γίνει στα σχόλια που αφορούν τον πρόλογο τα οποία πρέπει να εμφανίζονται στην αρχή κάθε προγραμματιστικής ενότητας. Ένας πρόλογος θα πρέπει να περιέχει τις εξής πληροφορίες:

Μία μικρή περίληψη για το τι κάνει το πρόγραμμα και τι μεθόδους χρησιμοποιεί.

- Τα ονόματα των προγραμματιστών.
- Την ημερομηνία ολοκλήρωσης του προγράμματος.
- Μία αναφορά στα τεχνικά βιβλία και βοηθήματα που μπορούν να δώσουν περισσότερες πληροφορίες για το πρόγραμμα.
- Μια μικρή αναφορά σε όλες τις τροποποιήσεις που έγιναν στο πρόγραμμα και ήταν αναγκαίες.

Όπως:

```
/*    Programmer Karandreas Vasilis
      Completion Date 28-10-2013
      Book used to help create the program Theory of Numbers by P.Tsagkaris
      I had to modify the program for odd numbers to run a bit faster
      ( now using step 2)        */
```

Αν η προγραμματιστική ενότητα αφορά μία συνάρτηση τότε ο πρόλογος θα πρέπει οπωσδήποτε να περιέχει τα εξής δύο σημεία:

- Τις συνθήκες εισαγωγής δεδομένων. Η αρχική κατάσταση της συνάρτησης και οι αρχικές τιμές που του δίνονται.
- Τις καταστάσεις εξόδου. Τι τιμές επιστρέφει η συνάρτηση μετά την εκτέλεση της.

Επίσης η επεξήγηση κάθε ξεχωριστού στοιχείου του προγράμματος όπως οι μεταβλητές και οι σταθερές είναι επίσης πολύ σημαντικό. Το πρόγραμμα στο τέλος του κεφαλαίου έχει τέτοιου είδους σχόλια για να κοιτάξετε.

Ο διαχωρισμός ενός προγράμματος σε ενότητες, χρησιμοποιώντας είτε σχόλια είτε κενές γραμμές είναι απαραίτητος. Παράδειγμα:

```
#include <stdio.h>
#include <math.h>

#define MAXNUM 400000

int calculate_sum_of_divisors(int num);

int Sum_of_divisors ;
```

Η χρησιμοποίηση σχολίων θα πρέπει να γίνεται είτε με **//** για σχόλια γραμμής (οι πιο παλιοί μεταγλωττιστές δεν το έχουν) είτε με **/* */** για σχόλια που επεκτείνονται και στην επόμενη γραμμή. Γενικά είναι καλό να προτιμάμε την δεύτερη περίπτωση ακόμα και όταν τα σχόλια περιορίζονται σε μια γραμμή. Παράδειγμα:

```
/* floor rounds to largest integral value not greater than square root of num */
```

Τέλος εξίσου σημαντικά είναι για τους προγραμματιστές να:

- σχολιάζουν εντολές που δεν είναι προφανείς.
- συντονίζουν τα σχόλια με πιθανές αλλαγές στο πρόγραμμα.
- σχολιάζουν συναρτήσεις και καθολικά δεδομένα [5].

Όσον αφορά το σχολιασμό του προγράμματος θα πρέπει να αποφεύγουν να:

- τονίζουν απλώς τον σκοπό μιας έκφρασης που έτσι και αλλιώς είναι προφανής.
- χρησιμοποιούν κρυπτογραφημένα σχόλια.
- να σχολιάζουν κακό κώδικα αντί να τον ξαναγράφουν [6].

2.4 Οριζόντια διαστήματα (Indentation) και γενική μορφή προγραμμάτων

Τα προγράμματα στην C αποτελούν μια σειρά από εμφωλευμένες δομές (nested structures). Το μέγεθος ή βάθος της διαπλοκής καλείται επίπεδο μιας εντολής. Αρχικά όλα τα προγράμματα βρίσκονται στο επίπεδο 0. Κάθε φορά που εισάγουμε ένα βρόχο ή μία υποθετική εντολή τότε πηγαίνουμε στο επόμενο επίπεδο. Η τακτική που ακολουθούμε για τα οριζόντια διαστήματα είναι να ευθυγραμμίζουμε όλες εκείνες τις εντολές που βρίσκονται στο ίδιο επίπεδο έτσι ώστε να μπορούμε εύκολα να ξεχωρίζουμε την εμφωλευμένη δομή ενός προγράμματος. Παράδειγμα:

```
if( num % 2 != 0 ){
```

```
    /* We check with step 2 and we omit the even numbers */
    for( i = 1; i <= root; i = i + 2){
        /* If the number divides num we add him and their pair
         (num / i) to the Sum_of_divisors */
        if( num % i == 0 ){
            Sum_of_divisors = ( Sum_of_divisors + i + ( num / i )) ;
        }
    }
}
```

Εδώ θα πρέπει να τονίσουμε ότι οι κανόνες στην τοποθέτηση οριζόντιων διαστημάτων δεν δημιουργούν μια καινούργια δομή αλλά τονίζουν την ήδη υπάρχουσα.

Δύο άλλες οδηγίες για τα οριζόντια διαστήματα, μικρότερης σημασίας, αφορούν τη τακτική που ακολουθούμε για τις περιπτώσεις πολλών εντολών σε μία γραμμή και μίας εντολής σε πολλές γραμμές.

Η πρώτη περίπτωση πρέπει γενικά να αποφεύγεται παρόλο που η δομή της C επιτρέπει την τοποθέτηση πολλών εντολών σε μία γραμμή. Η τοποθέτηση πολλών εντολών σε μία γραμμή αφενός μας δημιουργεί προβλήματα στην ανάγνωση ενός προγράμματος γιατί γίνεται εύκολο να χάσουμε μια εντολή και αφετέρου δεν τονίζει την εμφωλευμένη δομή του προγράμματος για τα διάφορα επίπεδα.

Στην περίπτωση που η εντολή είναι πολύ μεγάλη και δεν χωράει σε μία γραμμή θα πρέπει να την σπάσουμε σε περισσότερες γραμμές έτσι ώστε δομικά κομμάτια της εντολής που σχετίζονται, να είναι ευθυγραμμισμένα. Παράδειγμα:

```
printf("The sum of the divisors of the number %d equals to %d \n",
       currentnum,Sum_of_divisors);
```

και όχι

```
printf("The sum of the divisors of the number %d equals to %d \n",
currentnum,Sum_of_divisors);
```

Τέλος είναι επιθυμητό να τονίζουμε με παρενθέσεις τις προθέσεις μας. Παράδειγμα

`Sum_of_divisors = (Sum_of_divisors + i + (num / i));` και όχι

`Sum_of_divisors = (Sum_of_divisors + i + num / i);`

2.5 Επίλογος

Σε αυτό το κεφάλαιο συζητήσαμε για τους κύριους προβληματισμούς στην μορφή προγραμματισμού όπως ονοματολογία, διαύγεια στην έκφραση και σχόλια. Είναι δύσκολο να διαφωνήσουμε ότι αυτά τα πράγματα είναι κακά.

Αλλά γιατί να μας ανησυχεί η μορφή; Ποιος ενδιαφέρεται πως ένα πρόγραμμα μοιάζει αν δουλεύει; Δεν χρειάζεται πάρα πολύ χρόνο για να φαίνεται όμορφο; Δεν είναι οι κανόνες αυθαίρετοι ούτως ή άλλως;

Η απάντηση είναι ότι ο καλογραμμένος κώδικας είναι ευκολότερος να διαβαστεί και να κατανοηθεί, σχεδόν σίγουρα έχει λιγότερα λάθη και το πιο πιθανό να είναι πιο μικρός σε σχέση με έναν κώδικα που απρόσεκτα έχει πεταχτεί μαζί και ποτέ δεν έχει γυαλιστεί. Στην βιασύνη να τελειώσουμε τα προγράμματα εξ' αιτίας κάποιας προθεσμίας, είναι εύκολο να παραμερίσουμε την μορφή. Αυτή μπορεί να είναι μια απόφαση που θα μας στοιχίσει πολύ σε χρόνο. Η κύρια παρατήρηση είναι ότι η καλή μορφή πρέπει να είναι θέμα συνήθειας. Αν σκέφτεσαι την μορφή καθώς γράφεις κώδικα και αν αφιερώσεις χρόνο να την ξαναδείς και την βελτιώσεις θα αναπτύξεις καλές συνήθειες. Όταν γίνουν

αυτόματες το υποσυνείδητο σου θα αναλάβει πολλές από τις μικρές λεπτομέρειες και ο κώδικας που θα παράγεις κάτω από πίεση θα είναι καλύτερος [7].

2.6 Κώδικας

Ακολουθεί ένα παράδειγμα κώδικα βασισμένο στις παραπάνω οδηγίες.

```
/*    Programmer Karandreas Vasilis
      Completion Date 28-10-2013
      Book used to help create the program Theory of Numbers by P.Tsagkaris
      I had to modify the program for odd numbers to run a bit faster
      ( now using step 2)        */

#include <stdio.h>
#include <math.h>

#define MAXNUM 400000

int calculate_sum_of_divisors(int num);

int Sum_of_divisors ;

int main()
{
      int  currentnum ;

      /* we check the numbers from 2 to MAXNUM */
      for( currentnum = 2; currentnum < MAXNUM; currentnum++){
            Sum_of_divisors = 0;

            calculate_sum_of_divisors(currentnum);

            printf("The sum of the divisors of the number %d equals to %d \n",
                  currentnum,Sum_of_divisors);
      }

      return 0;
}

/* This function takes integers as input (it has no initial state) and it returns the sum of its
divisors as output with the command return */
int calculate_sum_of_divisors( int num )
{
      /* root is the root of number we have an input */
      int root, i;

      /* floor rounds to largest integral value not greater than square root of num */
      root = floor( sqrt(( double ) num ));
      /* case that num is a perfect square */
      if( root * root == num ){
            Sum_of_divisors = Sum_of_divisors + root ;
            /* we don't want to check till r in the next steps cause we will take
              the root 2 times in the Sum_of_divisors*/
```

```
            root = root - 1 ;
        }
        /* number is odd */
        if( num % 2 != 0 ){
            /* We check with step 2 and we omit the even numbers */
            for( i = 1; i <= root; i = i + 2 ){
                /* If the number divides num we add him and their pair
                (num / i) to the Sum_of_divisors */
                if( num % i == 0 ){
                    Sum_of_divisors = ( Sum_of_divisors + i + ( num / i )) ;
                }
            }
        }
        /* number is even */
        else
        /* We check  till the square root to see which number divide num */
            for( i = 1; i <= root; i++){
                if( num % i == 0){
                    /* and then we add the number and their pair
                    ( num / i )to the Sum_of_divisors */
                    Sum_of_divisors = (Sum_of_divisors + i +  ( num / i ));
                }
            }

return Sum_of_divisors;
}
```

3. ΣΥΜΠΕΡΙΦΟΡΑ ΠΡΟΓΡΑΜΜΑΤΩΝ ΚΑΤΑ ΤΗΝ ΕΚΤΕΛΕΣΗ ΤΟΥΣ

3.1 Εισαγωγή

Στο κεφάλαιο 2 αναπτύξαμε όλες εκείνες τις οδηγίες που αφορούν την ανάπτυξη ευκρινών και ευανάγνωστων προγραμμάτων. Η ευκρίνεια δεν είναι όμως το μόνο που μας απασχολεί σε ένα πρόγραμμα. Για να μεγιστοποιήσουμε την χρησιμότητα ενός προγράμματος θα πρέπει επίσης τα χαρακτηριστικά του να ακολουθούν κάποιες δεσμεύσεις και περιορισμούς. Σε ένα πρόγραμμα δεν μας ενδιαφέρει μόνο η ορθότητα, αλλά και τα χαρακτηριστικά που αφορούν περισσότερο την εκτέλεση του. Μερικά από αυτά θα τα δούμε στα επόμενα κεφάλαια.

3.2 Ανθεκτικότητα (Robustness)

Ένα ανθεκτικό πρόγραμμα δίνει λογικά αποτελέσματα για ένα οποιοδήποτε σύνολο δεδομένων ανεξάρτητα από το πόσο ακατάλληλα και ασύμβατα είναι αυτά. Είναι σκόπιμο να τονίσουμε εδώ πως λογικά αποτελέσματα δεν σημαίνει και σωστά. Για πολλά προβλήματα θα υπάρχουν σύνολα δεδομένων για τα οποία δεν θα μπορούμε να δώσουμε μια απάντηση ή να εφαρμόσουμε κάποιο αλγόριθμο. Μπορούμε όμως πάντοτε να εκτελέσουμε κάποιες λειτουργίες οι οποίες θα βοηθούν τον χρήστη. Αυτές θα μπορούσαν να αφορούν την εκτύπωση κάποιου κατάλληλου μηνύματος, την εξήγηση γιατί κάποιες λειτουργίες δεν μπορούν να εκτελεστούν, ή αν είναι δυνατό την ανατροφοδότηση του χρήστη με κάποιες οδηγίες που θα αφορούν την διόρθωση των δεδομένων εισαγωγής, πριν αυτά επανεισαχθούν στο πρόγραμμα.

Γενικά οι κανόνες ανθεκτικότητας αφορούν όλες εκείνες τις οδηγίες που απαιτούνται έτσι ώστε σε ένα πρόγραμμα να μπορεί να τερματίσει πάντα φυσιολογικά την εκτέλεση του.

3.2.1 Έλεγχος δεδομένων εισόδου

Η πιο σημαντική λειτουργία που εκτελεί ένας προγραμματιστής έτσι ώστε να εξασφαλίσει την ανθεκτικότητα στο πρόγραμμα του είναι ο πλήρης έλεγχος όλων των στοιχείων εισόδου. Τα πιο συνηθισμένα λάθη στην εισαγωγή δεδομένων είναι:

- Λάθος πληκτρολόγηση (π.χ. 7 αντί για 1).
- Λανθασμένη σειρά εισαγωγής των στοιχείων εισόδου.
- Μη κατανόηση της μορφής εισόδου των δεδομένων (π.χ. ημερομηνίας ως 10 Ιουνίου ή 6/10 ή 161(Οι μέρες που έχουν περάσει από την 1η Ιανουαρίου).

Αν ακατάλληλα στοιχεία εισαχθούν στο πρόγραμμα τότε το πρόγραμμα μπορεί να εκτελέσει απρόβλεπτες λειτουργίες ανεξάρτητα από το πόσο καλά είναι γραμμένο. Για αυτό τα δεδομένα εισόδου θα πρέπει να ελέγχονται πρώτα αν είναι τυπικά σωστά. Πρέπει πάντα να βρίσκονται μέσα στα όρια που θέτουν το ίδιο το πρόβλημα αλλά και η πραγματικότητα. Για παράδειγμα οι προδιαγραφές ενός προγράμματος μισθοδοσίας μπορεί να θέτει ορισμένα όρια όπως:

0 < ID number<=99999

0 <= Dept number<=99

1,90 <=Payrate

0 <= Exemptions

Μπορούμε επιπλέον να βάλουμε και ορισμένα φυσικά όρια :

0 <= Daily Hours Worked <= 24

0 <= Days Worked This Week <= 7

και να ελέγξουμε αν τα στοιχεία εισόδου για την μισθοδοσία είναι μέσα στα όρια:

```
#define iderror -1
#define deperror -2
#define payerror -3
#define exemerror -4
#define ehourserror -5
#define dayerror -6

int error = 0;

if (id <= 0 || id > 99999)
      error = error + iderror;
if (dept < 0 || dept > 99)
      error = error + depterror;
if (payrate < 1,90)
      error = error + payerror;
if (exemptions < 0)
      error = error + exemerror;
if (hours < 0 || hours > 24)
      error = error + hourserror;
if (days < 0 || days > 7)
      error = error + dayerror;
if (error)
      /* process the data */
else
      /* invoke error function */
```

Από το παραπάνω παράδειγμα είναι φανερό πως ένα άλλο σημαντικό στοιχείο είναι ότι σε περίπτωση λάθους θα πρέπει να είμαστε σε θέση να πούμε στον χρήστη τι συνέβη ακριβώς. Επίσης θα πρέπει να γίνεται έλεγχος όχι μόνο για τις τυπικές τιμές αλλά και για τις πιθανές. Για παράδειγμα μία ιδιαίτερα υψηλή τιμή που θα αφορούσε μία μεταβλητή στην οποία θα καταχωρούνταν ο αριθμός των μελών μιας οικογένειας θα πρέπει να απορριφθεί. Ακόμη μία ωριαία αποζημίωση της τάξης των 300 ευρώ. θα έπρεπε να μας προβληματίσει. Μπορούμε με ειδικές εντολές να επιβεβαιώσουμε ότι όλες οι τιμές των δεδομένων είναι όχι μόνο οι τυπικά αποδεκτές αλλά και οι πιθανές.

```
/*dont accept pay rate over 300*/
if( payrate > 300 )
      error = error + overpayerror;
/*do not accept 20 or more dependents*/
if( exemptions >= 20 )
      error = error + over_exempt_error;
```

Σε αυτές τις περιπτώσεις είναι θέμα του χρήστη να αποφασίσει για την ορθότητα ή όχι των δεδομένων εισόδου εφόσον τυπικά οι τιμές δεν μπορούν να θεωρηθούν λάθος. Από την πλευρά του προγράμματος μια προειδοποίηση είναι αρκετή. Μία άλλη ασφαλιστική δικλείδα για τον έλεγχο των δεδομένων εισόδου είναι η άμεση εκτύπωση των δεδομένων εισόδου (echo printing) με την χρήση κάποιου διευκρινιστικού επιθέματος.

```
scanf("%d ",&x);
```

```
printf(" x equals to %d ",x);
```

Θα πρέπει να τονίσουμε ότι παρ' όλο που ο έλεγχος των δεδομένων απαιτεί το 20-40 % των εντολών ενός προγράμματος, το πρόγραμμα πρέπει να προστατεύεται από όλα τα ακατάλληλα δεδομένα όσο ασύμβατα και αν είναι. Ο αριθμός των γραμμών που χρειάζεται για να πετύχετε το επίπεδο ασφάλειας που θέλετε δεν έχει τόσο μεγάλη σημασία μπροστά στην προστασία που θα πετύχετε.

3.2.2 Προστασία από λάθη κατά την διάρκεια της εκτέλεσης (Run-time errors)

Ένα λάθος εκτέλεσης κάνει ένα πρόγραμμα να σταματήσει ξαφνικά χωρίς να δώσει αποτελέσματα. Αυτό μπορεί να συμβεί είτε από ακατάλληλα δεδομένα εισόδου ή ακόμη και από δεδομένα έγκυρα τα οποία όμως λόγω της ειδικής τιμής τους δημιουργούν πρόβλημα σε κάποιες λειτουργίες εκτέλεσης ενός προγράμματος. Για παράδειγμα ακόμη και αν έχουμε περιορίσει την τιμή μιας μεταβλητής που αναπαριστάνει μια γωνία στις τιμές: $0^O <= \theta <= 360^O$ η εκτέλεση μιας λειτουργίας στο πρόγραμμα που θα περιέχει την εφ(θ) για τιμές της $\theta = 90^O$ και 270^O θα αναγκάσει το πρόγραμμα να σταματήσει.

Στην C οι πιο συνηθισμένοι λόγοι για τους οποίους συμβαίνουν λάθη εκτέλεσης είναι:

- Υπερχείλιση: Αριθμητικές πράξεις που παράγουν αποτέλεσμα πολύ μεγάλο για να αναπαρασταθεί.
- Διαίρεση με το μηδέν: Να διαιρέσεις μια αριθμητική αξία με το μηδέν.
- Άκυρη μετατόπιση(shift): Το να μετατοπίσεις μια ακέραια τιμή κατά ένα ποσό το οποίο παράγει ένα απροσδιόριστο αποτέλεσμα σύμφωνα με το πρότυπο C.
- Σφάλματα μνήμης: Πρόσβαση σε μια άκυρη περιοχή μνήμης με έναν τρόπο που παράγει ένα απροσδιόριστο αποτέλεσμα, όπως η πρόσβαση σε έναν δείκτη που βρίσκεται εκτός ορίων.
- Πρόσβαση σε μη αρχικοποιημένα δεδομένα: Πρόσβαση στην μνήμη πριν αυτή αρχικοποιηθεί, έτσι το αποτέλεσμα της πρόσβασης είναι απροσδιόριστο.

Για να αποτρέψουμε την εμφάνιση τέτοιων καταστάσεων πρέπει πριν από την εκτέλεση ενός προγράμματος να κάνουμε αρκετούς ελέγχους. Η πρόληψη τέτοιων καταστάσεων ονομάζεται αμυντικός προγραμματισμός.

Παράδειγμα σφάλματος μνήμης, η εκτέλεση του κώδικα που φαίνεται παρακάτω μπορεί να αναγκάσει ένα πρόγραμμα που τον περιλαμβάνει να τερματίσει βιαίως εφ' όσον η ποσότητα sqr(b) - 4.0 * a * c γίνει αρνητική :

```
root = ( -b + sqrt( b * b )-( 4 * a * c ))/( 2 * a);
```

αντί για αυτού καλύτερα να γράψουμε:

```
if(discriminant>=0) {
    /* make sure numbers aren't int so as the result of the division isn't int */
    root1 = ( -b + sqrt(disriminant)) / (2 * a);
    root2 = (-b  - sqrt(disriminant)) / (2 * a);
}
else
    /*function to compute complex roots*/
    compute_complex_roots( a, b, c);
```

Ένα ακόμα παράδειγμα σφάλματος μνήμης είναι το:

```
int main()
{
    int i = 0, n = 8;
    int a[5];

    /* in this for we exceed the dimension of the array */
    for( i = 0; i < n; i++) {
        a[i] = a[i] + 3;
    }
}
```

Εδώ θα έχουμε δείκτη που βρίσκεται εκτός ορίων.

Παράδειγμα πρόσβασης σε μη αρχικοποιημένα δεδομένα

```
int sum( int * A, int n)
{
    int x , i = 0;

    while( i < n) {
        x = x + A[i];
        i++;
    }
    return x;
}
```

Σε αυτήν την περίπτωση το x δεν έχει αρχικοποιηθεί.

3.2.3 Προστασία από λάθη απεικόνισης

Σε οποιαδήποτε μηχανή μπορούμε να απεικονίσουμε ακριβώς οποιοδήποτε βαθμωτό τύπο δεδομένων θέλουμε, εκτός από τους πραγματικούς. Γι' αυτό όταν γράφουμε προγράμματα θα πρέπει να αποφεύγουμε να εξετάζουμε την ισότητα μεταξύ στοιχείων που είναι πραγματικού τύπου δεδομένων. Θα πρέπει είτε να χρησιμοποιούμε ένα συνηθισμένο τύπο ο οποίος δεν θα επηρεάζεται από λάθη στρογγυλοποίησης είτε να γράφουμε τα προγράμματα μας με τέτοιο τρόπο ώστε να μην είναι ευαίσθητα σε λάθη προσέγγισης που αφορούν πραγματικούς αριθμούς. Για παράδειγμα το παρακάτω κομμάτι κώδικα έχει σκοπό να υπολογίσει τις τιμές του x στο διάστημα από 0.0 ως 2.0 σε βήματα των 0.2, με συνολικά 11 φορές.

```
x = 0;

while( x != 2) {
    ...
    /*process this value of x*/
    x = x + 0.2;
}
```

Η εκτέλεση όμως του κώδικα αυτού μπορεί να οδηγήσει το πρόγραμμα σε ατέρμονα βρόχο. Εξαιτίας των λαθών στρογγυλοποίησης (ειδικά σε παλιούς υπολογιστές και εκδόσεις της C) η τιμή του x μετά από την 11η εκτέλεση του βρόχου μπορεί να μην είναι ακριβώς 2.2 αλλά 2.2-ε όπου ε είναι κάποια μικρή θετική τιμή. Η μικρή αυτή διαφορά θα κάνει την ανισότητα να παραμένει αληθής με αποτέλεσμα ο βρόχος να εκτελείται συνεχώς. Μια καλύτερη μέθοδος για την δόμηση του παραπάνω κώδικα θα ήταν η εξής:

```
x = 0.0;

while( x <= 2.0) {
      /*process this value of x*/
     x = x + 0.2;
}
```

Τώρα ο βρόχος θα εκτελεστεί ακριβώς 11 φορές.

Μια άλλη πηγή λαθών είναι τα λάθη προσέγγισης. Αυτά δημιουργούνται από την προσέγγιση μιας ατέρμονης μαθηματικής διαδικασίας με ένα περιορισμένο αριθμό βημάτων. Στην περίπτωση αυτή εν γνώση μας παίρνουμε κάποια αποτελέσματα που περιέχουν λάθη.

3.2.4 Ευχάριστο τελείωμα

Το χειρότερο πράγμα με τα λάθη εκτέλεσης είναι ότι κάνουν το πρόγραμμα να χάσει τον έλεγχο της επεξεργασίας των δεδομένων. Συνήθως η δημιουργία ενός λάθους εκτέλεσης συνοδεύεται και από ένα κρυπτογραφημένο μήνυμα που δεν βοηθάει καθόλου τον χρήστη.

Ελέγχοντας όμως εμείς όλες τις συνθήκες που οδηγούν σε λάθη εκτέλεσης, μπορούμε να αποφασίσουμε εύκολα για το τι μέτρα πρέπει να πάρουμε και όχι να αφήσουμε το σύστημα να πάρει αυτόματα μόνο του τα αναγκαία μέτρα. Ένα πρόγραμμα θα πρέπει εκτός από το να δίνει μηνύματα με νόημα στον χρήστη, να εφαρμόζει και τους ακόλουθους κανόνες με την ακόλουθη σειρά:

1. Να κάνει μια λογική υπόθεση για το λανθασμένο στοιχείο η οποία να επιτρέπει την συνέχιση της επεξεργασίας αυτού του στοιχείου. Κατόπιν να ενημερώνει τον χρήστη για αυτή την υπόθεση.
2. Να απορρίπτει το λανθασμένο στοιχείο χωρίς να σταματάει, αν αυτό είναι Δυνατό και να επεξεργάζεται τα υπόλοιπα στοιχεία της ίδιας εγγραφής (record).
3. Να απορρίπτει όλη την εγγραφή που επεξεργάζεται και που περιέχει το λανθασμένο στοιχείο και να συνεχίσει να επεξεργάζεται το υπόλοιπο αρχείο μέχρι να φτάσει στο τέλος του (until end of file ή eof).
4. Να απορρίψει όλο το αρχείο που περιέχει το λανθασμένο στοιχείο αλλά να συνεχίσει να επεξεργάζεται αν υπάρχουν τα επόμενα αρχεία μέχρι το τέλος όλης της πληροφορίας που υπάρχει.
5. Εφόσον μπορούμε να εφαρμόσουμε κάποιο από τους παραπάνω κανόνες φτάνουμε στον κανόνα 5. Τελειώνουμε το πρόγραμμα μας με χάρη. Δημιουργούμε χρήσιμα μηνύματα που θα επιτρέψουν στον χρήστη να ξανατρέξει το πρόγραμμα.

3.3 Σωστή Χρήση των Συναρτήσεων

Η σωστή χρήση υποπρογραμμάτων όπως είναι οι συναρτήσεις είναι σημαντική για την ανάπτυξη προγραμμάτων οποιουδήποτε μεγέθους και πολυπλοκότητας. Σε αυτό το κεφάλαιο θα υποδείξουμε μερικές οδηγίες μορφοποίησης για την δημιουργία εύχρηστων και αποδοτικών υποπρογραμμάτων.

3.3.1 Τρόποι Δηλώσεων Συναρτήσεων

Υπάρχουν δύο τρόποι για να δηλώσεις μια συνάρτηση μέσα σε ένα πρόγραμμα. Ο πρώτος είναι να γράψεις όλη την συνάρτηση στην αρχή όπως στο παρακάτω παράδειγμα:

```
#include<stdio.h>

int sum( int n1,int n2 ) {
    return( n1 + n2 );
}

int main()
{
    int num1 = 11, num2 = 22;
    int result;

    result = sum( num1, num2 );

    printf("\nResult : %d",result);

    return();
}
```

και ο δεύτερος να δηλώσεις την συνάρτηση στην αρχή και μετά να την υλοποιήσεις:

```
#include<stdio.h>

int sum( int n1,int n2 );

int main()
{
    int num1 = 11,num2 = 22;
    int result;

    result = sum( num1, num2 );

    printf("\nResult : %d",result);

    return();
}

int sum( int n1, int n2 ) {
    return( n1 + n2 );
}
```

Σε κάθε περίπτωση μια συνάρτηση δεν επιτρέπεται να χωρίζεται σε διαφορετικά αρχεία.

3.3.2 Τρόποι εισαγωγής παραμέτρων

Στην C όλα τα ορίσματα των συναρτήσεων μεταβιβάζονται κατά αξία (by value). Αυτό σημαίνει ότι η καλούμενη συνάρτηση παίρνει τις τιμές των ορισμάτων της σε προσωρινές μεταβλητές και όχι στις πρωτότυπες. Άρα στην C η καλούμενη συνάρτηση δεν μπορεί να αλλάξει άμεσα μια μεταβλητή της καλούσας συνάρτησης, μπορεί να αλλάξει μόνο το δικό της προσωρινό αντίγραφο [8].

```
#include < stdio.h >

int main()
```

```
{
    int x, y, temp;

    printf("Enter the value of x and y\n");
    scanf("%d %d", & x, & y);
    printf("Before Swapping\n x = %d\n y = %d \n", x, y);

    temp = x;
    x = y;
    y = temp;

    printf("After Swapping\nx = %d\ny = %d\n", x, y);

    return 0;
}
```

Αν για οποιοδήποτε λόγο θέλουμε να αλλάξουμε τις μεταβλητές τότε θα πρέπει να χρησιμοποιήσουμε δείκτες.

```
#include <stdio.h>

void swap(int*, int*);

int main()
{
    int x, y;

    printf("Enter the value of x and y\n");
    scanf("%d%d",&x,&y);
    printf("Before Swapping\nx = %d\ny = %d\n", x, y);

    swap(&x, &y);

    printf("After Swapping\nx = %d\ny = %d\n", x, y);

    return 0;
}

void swap(int *a, int *b) {
    int temp;

    temp = *b;
    *b   = *a;
    *a   = temp;
}
```

Γενικά η κλίση κατά αξία είναι πλεονέκτημα γιατί συνήθως οδηγεί σε πιο συμπαγή προγράμματα με λιγότερες παραπανίσιες μεταβλητές, γιατί οι παράμετροι μπορούν να αντιμετωπίζονται στην καλούμενη ρουτίνα σαν τοπικές μεταβλητές και να παίρνουν κατάλληλες αρχικές τιμές [9].

3.3.3 Εμβέλεια και χρόνος ζωής μεταβλητών

Οι μεταβλητές που ορίζονται μέσα σε μία συνάρτηση λέγονται αυτόματες ή τοπικές. Ο λόγος είναι ότι οι μεταβλητές αυτές έχουν περιορισμένη εμβέλεια και χρόνο ζωής. Ισχύουν μόνο μέσα στη συνάρτηση που έχουν ορισθεί και για όσο χρόνο εκτελείται η συνάρτηση.

Εκτός από τις αυτόματες μεταβλητές, μπορούμε να ορίσουμε σε ένα πρόγραμμα C και εξωτερικές ή καθολικές μεταβλητές. Οι μεταβλητές αυτές ορίζονται όπως και οι αυτόματες (με τον τύπο τους, το όνομά τους και με πιθανή αρχικοποίηση), μόνο που ορίζονται έξω από οποιαδήποτε συνάρτηση, και έτσι είναι διαθέσιμες σε πολλές συναρτήσεις. Οι εξωτερικές μεταβλητές πρέπει να ορίζονται μόνο μια φορά. Οι εξωτερικές μεταβλητές έχουν το προτέρημα ότι είναι ορατές από κάθε συνάρτηση, μπορούν δηλαδή να χρησιμοποιηθούν στα σώματα των συναρτήσεων, και διατηρούνται καθ' όλη τη διάρκεια της εκτέλεσης του προγράμματος. Μέσω των εξωτερικών μεταβλητών, μπορεί να επιτευχθεί επικοινωνία μεταξύ συναρτήσεων [10]. Έτσι δεν χρειάζεται να χρησιμοποιήσουμε επιπλέον παραμέτρους στην κλήση των συναρτήσεων για να περάσουμε δεδομένα από την μία στην άλλη. Παράδειγμα:

```
#include <stdio.h>

/* global variable declaration */
int G;

int main ()
{
    /* local variable declaration */
    int a, b;

    /* actual initialization */
    a = 10;
    b = 20;

    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

    return 0 ;
}
```

Μη αρχικοποιημένες εξωτερικές μεταβλητές θεωρούνται ότι έχουν τιμή 0. Μη αρχικοποιημένες αυτόματες μεταβλητές έχουν απροσδιόριστες τιμές [11]. Σε ακραία περίπτωση θα μπορούσαμε να απαλείψουμε όλες τις τοπικές μεταβλητές και να έχουμε μόνο καθολικές αλλά τέτοια προγράμματα είναι δυσανάγνωστα και δύσκολα διατηρήσιμα. Αν ένα πρόγραμμα είναι διασπασμένο σε πολλά αρχεία και θέλουμε να έχουμε μία εξωτερική μεταβλητή που να είναι ορατή από τις συναρτήσεις σε όλα τα αρχεία, τότε πρέπει σε ακριβώς ένα αρχείο να δώσουμε τον ορισμό της μεταβλητής.
Για παράδειγμα:

```
int Global_id;
```

Ενώ σε καθένα από τα άλλα αρχεία πρέπει να κάνουμε μία δήλωση της μεταβλητής, προτάσσοντας τον προσδιοριστή extern. Για παράδειγμα:

```
extern int Global_id;
```

Αν θέλουμε η εμβέλεια μίας εξωτερικής μεταβλητής να είναι μόνο το αρχείο στο οποίο ορίζεται και να μην συγχέεται με κάποια εξωτερική μεταβλητή με το ίδιο όνομα, αλλά σε

άλλο αρχείο, μπορούμε να προτάξουμε στον ορισμό της τον προσδιοριστή static [12]. Για παράδειγμα:

```
static double tax_percentage;
```

Αν υπάρχει σύγκρουση ονόματος μεταξύ μίας καθολικής μεταβλητής και μίας τοπικής μέσα σ΄ ένα μπλοκ, ισχύει η αυτόματη [13]. Παράδειγμα:

```
#include <stdio.h>

/* global variable declaration */
int n = 0;

int main ()
{
    /* local variable declaration */
    int i = 0, n = 5;

    for( i = 0, i < n, i++ ) {
        …
    }

    return 0;
}
```

Το n θα είναι 5 μέσα στην **for**.

3.3.4 Σημαίνουσες σημαίες (signal flags)

Μία από τις πιο σημαντικές τιμές που πρέπει να υπολογίζει μία συνάρτηση είναι η σημαίνουσα σημαία. Αυτή είναι μία σημαντική παράμετρος που προσδιορίζει αν μία συνάρτηση εκτέλεσε τις εντολές που έπρεπε και αν όχι ποια ειδική περίπτωση εμπόδισε την εκτέλεση τους. Η παράμετρος σημαίας κατόπιν εξετάζεται από το κύριο πρόγραμμα για να προσδιορίσει το αποτέλεσμα της εκτέλεσης της συνάρτηση πριν χρησιμοποιήσει τα δεδομένα εξόδου της που μπορεί να είναι και λάθος. Παράδειγμα:

```
int rowsum(int size, int row, int sum, int **a , int *success,) {
    int i;

    /* if we are inside the array then we have success */
    if( row >= 1 && row <= size) {
        *success = 1;
        sum = 0;

        for( i = 0; i < size; i++){
            sum = sum + a[row,i];
        }
    }
     else {
        *success = 0;
    }
    return 0;

}
```

Οποιοδήποτε πρόγραμμα καλέσει την παραπάνω συνάρτηση θα πρέπει πρώτα να ελέγξει την σημαία εξόδου της πριν χρησιμοποιήσει τα δεδομένα εξόδου της για να συνεχίσει την ροή εκτέλεσης του προγράμματος:

```
rowsum(size, row, sum, success, a[x,y]);

if(success==1) {
    /*carry on with computation*/
    ...
}
else {
    /*invoke error recovery methods*/
    ...
}
```

Γενικά πάντως μία συνάρτηση δεν πρέπει να δίνει κατευθείαν αποτελέσματα εξόδου στο χρήστη με εντολές τύπου printf μέσα σε αυτή. Μία καλογραμμένη συνάρτηση θα πρέπει να κάνει τίποτε περισσότερο από τους υπολογισμούς για τους οποίους είναι γραμμένη και να γυρίζει στο κυρίως πρόγραμμα τα αποτελέσματα και την σημαία των υπολογισμών αυτών. Όλες οι άλλες αποφάσεις για το πως πρέπει να χρησιμοποιηθούν τα αποτελέσματα αυτά πρέπει να γίνονται έξω από αυτή.

3.4 Γενικότητα

Η γενικότητα ενός προγράμματος αποδεσμεύει την εκτέλεση του από ένα συγκεκριμένο σύνολο δεδομένων. Αυτό σημαίνει ότι το πρόγραμμα μπορεί να λειτουργεί με διαφορετικά σύνολα δεδομένων χωρίς να απαιτείται να επέμβουμε στον κώδικα.

Ένα παράδειγμα ενός μη γενικού κώδικα που υπολογίζει ένα συγκεκριμένο πλήθος στοιχείων των οποίων οι τιμές πρέπει να βρίσκονται μέσα σε συγκεκριμένα όρια τιμών φαίνεται παρακάτω:

```
int count = 0;
int sum = 0;
float mean;

for( i = 0; i < 25 ; i++) {
    if( score[i] >= 0 && score[i] <= 100) {
        count = count + 1;
        sum = sum + score[i];
    }
}
mean = sum / count;
```

Οποιαδήποτε αλλαγή στο πρόγραμμα θα απαιτούσε την αναζήτηση μέσα στον κώδικα:

- όλων εκείνων των σημείων που περιέχουν τις τιμές που αναφέρονται στο μέγιστο πλήθος των στοιχείων που προστίθενται (25).
- στο πάνω και κάτω όριο αυτών (0..100).
- την αλλαγή τους στις κατάλληλες τιμές.

Ακολουθώντας αυτή τη πρακτική η πιθανότητα να δημιουργήσουμε λάθη στον κώδικα είναι προφανώς πολύ μεγάλη.

Αν αντίθετα γράφαμε τον κώδικα με τέτοιο τρόπο έτσι ώστε τα όρια που χρησιμοποιούμε οπουδήποτε μέσα στο πρόγραμμα να είναι σταθερές τιμές που ορίζουμε στην αρχή του προγράμματος (#define) τότε θα είχαμε ένα πιο ευέλικτο

πρόγραμμα. Για παράδειγμα θα μπορούσαμε να γράφαμε το παραπάνω πρόγραμμα ως εξής:

```
#define datasets  25       /* number of data items */
#define  high  100         /* max range of valid scores */
#define low  0             /* min range of valid scores */

int count = 0;
int sum = 0;
float mean;

error = false;

for( i = 0; i < datasets; i++) {
     if( score[i] >= low && score[i] <= high) {
          count = count + 1;
          sum = sum + score[i];
     }
}
if( count > 0) {
     mean = sum / count;
}
else {
     printf(" error");
}
```

Τώρα είναι εύκολο να αλλάξουμε το πλήθος των επαναλήψεων και τα όρια του score αλλάζοντας μόνο μια τιμή στην αρχή του προγράμματος.

Η γενικότητα ενός προγράμματος αναπτύσσεται από τα πρώτα στάδια σχεδίασης του. Πρωταρχικός σκοπός ενός προγραμματιστή θα είναι η αύξηση της τόσο στα προγράμματα όσο και στις συναρτήσεις που αναπτύσσει.

3.5 Δυνατότητα μεταφοράς προγραμμάτων (Portability)

Γενικά είναι δύσκολο να γραφτεί λογισμικό που να τρέχει σωστά και αποδοτικά. Έτσι όταν ένα πρόγραμμα λειτουργεί σε ένα περιβάλλον, δεν θέλουμε να επαναλάβουμε όλη τη προσπάθεια αν χρειαστεί να το μετακινήσουμε σε άλλον μεταγλωττιστή ή επεξεργαστή ή λειτουργικό σύστημα. Ιδανικά θα θέλαμε να μην χρειαστεί οποιαδήποτε αλλαγή. Η ιδέα αυτή λέγεται δυνατότητα μεταφοράς προγραμμάτων.

Φυσικά ο βαθμός της μεταφοράς πρέπει να μετριαστεί από την πραγματικότητα. Δεν υπάρχει τέτοιο πράγμα σαν το απόλυτα μεταφερόμενο πρόγραμμα, μόνο ένα πρόγραμμα το οποίο δεν έχει δοκιμαστεί σε αρκετά περιβάλλοντα.

Οι πιο σημαντικές οδηγίες για να αυξηθεί ο βαθμός μεταφοράς είναι οι ακόλουθες:

- Μείνετε στην βασική έκδοση (standard). Το πρώτο βήμα για μεταφερόμενο κώδικα είναι φυσικά το πρόγραμμα να είναι σε γλώσσα υψηλού επιπέδου μέσα στη βασική έκδοση της γλώσσας. Τα δυαδικά δεν μεταφέρονται καλά, αλλά ο πρωτογενής κώδικας (source code) μεταφέρεται [14].
- Χρησιμοποιείστε τις βασικές βιβλιοθήκες. Η ίδια γενική συμβουλή ισχύει εδώ όπως και για τον πυρήνα της γλώσσας: μείνετε στην βασική έκδοση και μέσα στα παλιότερα και καλά καθιερωμένα συστατικά της [15].
- Χρησιμοποιείστε χαρακτηριστικά που είναι διαθέσιμα παντού. Η προσέγγιση

που προτείνουμε είναι αυτή της τομής: χρησιμοποιείστε χαρακτηριστικά που υπάρχουν σε όλα τα προοριζόμενα συστήματα, μην χρησιμοποιείτε ένα χαρακτηριστικό αν δεν υπάρχει παντού [16].

- Αποφύγετε την υπό συνθήκη μεταγλώττιση (conditional complilation). Η υπό συνθήκη μεταγλώττιση με #ifdef και παρόμοιες εντολές είναι δύσκολο να διαχειριστούν, γιατί οι πληροφορίες τείνουν να σκορπίζουν μέσα σε όλον τον κώδικα [17].
- Απομόνωση. Παρόλο που θα θέλαμε να έχουμε μόνο ένα αρχείο με πρωτογενή κώδικα που να μεταγλωττίζεται σε όλα τα συστήματα αυτό είναι μη ρεαλιστικό. Για αυτό όταν διαφορετικός κώδικας χρειάζεται για διαφορετικά συστήματα οι διαφορές είναι καλό να τοποθετούνται σε διαφορετικά αρχεία, ένα αρχείο για κάθε σύστημα [18].
- Οι αριθμοί απεικονίζονται με διαφορετικό τρόπο σε δυαδικό επίπεδο σε διαφορετικές αρχιτεκτονικές. Τα συστήματα Big-endian προτάσσουν το πιο σημαντικό byte πρώτο και τα συστήματα little-endian προτάσσουν το λιγότερο σημαντικό byte πρώτο [19].
- Μη υποθέτεις αγγλικά. Οι δημιουργοί των διεπαφών (interfaces) πρέπει να έχουν στον νου τους ότι διαφορετικές γλώσσες συχνά παίρνουν σημαντικά διαφορετικό αριθμό χαρακτήρων για να πουν το ίδιο πράγμα, έτσι πρέπει να υπάρχει αρκετός χώρος στην οθόνη και στους πίνακες, καθώς και ότι υπάρχει περίπτωση μη κατανόησης των μηνυμάτων [20].
- Μη υποθέτεις ascii, μπορεί να έχει χρησιμοποιηθεί άλλη απεικόνιση [21].
- Τέλος πρέπει να θυμάστε ότι το μέγεθος των τύπων δεδομένων ενδέχεται να είναι διαφορετικό [22].

3.6 Συμπεριφορά εισόδου-εξόδου

Οι περισσότερες από τις οδηγίες που έχουμε δώσει μέχρι τώρα αποσκοπούν στο να βοηθήσουν εκείνα τα άτομα που διαβάζουν, κατανοούν και τροποποιούν ένα πρόγραμμα.

Ακόμη τα περισσότερα από τα οποία έχουμε συζητήσει μέχρι τώρα δεν αφορούν τον τελικό χρήστη. Γι' αυτούς ένα πρόγραμμα είναι σαν ένα μαύρο κουτί. Ξέρουν μόνο τι μπαίνει και τι βγαίνει μέσα από αυτό.

Το μόνο που ενδιαφέρει πραγματικά τους χρήστες είναι η διεπαφή μεταξύ αυτών και του προγράμματος. Η καλαισθησία στην είσοδο και την έξοδο ενός προγράμματος είναι εξίσου σημαντική με την καλαισθησία και την σωστή δόμηση του κώδικα. Ειδικότερα όσον αφορά την συμπεριφορά ενός προγράμματος για την είσοδο και την έξοδο του ένας προγραμματιστής θα πρέπει να προσέχει να:

- δίνει πάντα στον χρήστη να καταλάβει τι τύπο δεδομένων θα πρέπει να εισάγει.
- αποφεύγει να γράφει προγράμματα που παγιδεύουν τον χρήστη σε ατέρμονους βρόχους σε περίπτωση λανθασμένων δεδομένων εισόδου, χωρίς να του προσφέρεται βοήθεια για να διορθώσει τυχόν λάθος του.
- δίνει στον χρήστη ένα μενού βοήθειας για να τον βοηθάει να προετοιμάζει τον τύπο των δεδομένων εισόδου και τον τρόπο που θα πρέπει να τα εισάγει.
- δίνει στον χρήστη την δυνατότητα να εισάγει τα δεδομένα με ένα φυσικό τρόπο όπως είναι η χρήση μιας φόρμας και όχι με τρόπο ευθύ και μηχανικό.
- χρησιμοποιεί σταθερές τιμές σε μερικά πεδία αν αυτές επαναλαμβάνονται συχνά για την μείωση του όγκου των δεδομένων εισόδου σε ένα πρόγραμμα.
- μην είναι επιφυλακτικός στο να σπαταλήσει αρκετό χρόνο έτσι ώστε τα δεδομένα εξόδου ενός προγράμματος να είναι καλαίσθητα, ευανάγνωστα και άμεσα αξιοποιήσιμα από τον τελικό χρήστη.

Παράδειγμα κώδικα:

```
/* This program calculates the nth root of a number */

#include <stdio.h>
#include  <math.h>
#include <stdlib.h>

main(int argc,char *argv[])
{
    int degree, num, ok = 0;
    float root;

/* we need to have exactly 3 arguments first the name of the program then the degree
of the root and then the number */
    if(argc!= 3) {
        printf("\nUsage: %s degree of root and number of whose root we want to
                calculate",argv[0]);
        exit(1);
    }

    degree = atoi(argv[1]);
    num = atoi(argv[2]);
    if( degree <= 2 || num <= 1) {
        printf("\nWrong values! degree must be higher than 2 and the number must be
                higher than 1.");
    }
    else {
        root = pow(num ,1./ degree);
    }

    if(degree == 3) {
        printf("\n The Third root of %d is %f ",num,root);
    }
    else{
        printf("\n The %d th root of %d is %f ",degree,num,root);
    }
}
```

3.7 Προχωρημένες Τεχνικές

Χρησιμοποιείται “καθαρή C”, δηλαδή κώδικα που να μπορεί να μεταγλωττιστεί από την C αλλά και από την C++. Να θυμάστε ότι ο μεταγλωττιστής στην C++ είναι πιο αυστηρός από αυτόν της C. Να αποφεύγετε λέξεις κλειδιά της C++ όπως bool, true, false [23].

1. Τα σχόλια μπορούν να δηλώνουν κώδικα που λείπει [24]. Παραδείγματος χάριν:

```
#include <stdio.h>

void swap(int*, int*);

int main()
{
    int x, y;
```

```
    printf("Enter the value of x and y\n");
    scanf("%d%d",&x,&y);
    printf("Before Swapping\nx = %d\ny = %d\n", x, y);

    /*function hasn't been written yet*/
    swap(&x, &y);

    printf("After Swapping\nx = %d\ny = %d\n", x, y);
    return 0;
}
```

2. Υπάρχουν κάποια πράγματα τα οποία αν και δεν είναι λάθη καλό θα ήταν να αποφεύγονται. Μερικά από αυτά είναι τα παρακάτω:

- Να προτιμάτε αρχικοποίηση από ανάθεση.
- Μη συγκρίνετε ρητά με τα true - false.
- Αποφύγετε την περιττή χρήση του σωστού / λάθους
- Κάντε πιο ξεκάθαρο τι επιστρέφει η κάθε συνάρτηση [25].

Θα δούμε ένα παράδειγμα κώδικα που θα έχει αυτά που πρέπει να αποφεύγουμε και θα ξαναγράψουμε τον κώδικα με το πώς θα ήταν καλό να γραφτούνε. Το πρόγραμμά μας θα παίρνει σαν είσοδο 8 αριθμούς και θα δίνει σαν έξοδο πρώτα όλους τους μονούς αριθμούς ταξινομημένους και μετά θα ακολουθούν οι ζυγοί αριθμοί ταξινομημένοι.

```
#include <stdio.h>
#include <stdbool.h>

/* function that decides if a number is even*/
bool is_even(int value);
/* function that decides if the matrix is sorted */
int is_sorted(int a[]);

int main()
{
    int array[8], i , swap;
    swap = 0;

    for ( i = 0 ; i <= 7; i++) {
        scanf("%d", &array[i]);
    }

    printf("The integers you put are:");

    for ( i = 0 ; i <= 7; i++){
        printf("  %d", array[i]);
    }
    /* while the numbers aren't sorted execute the while */
    while(is_sorted(array) == false) {
        /* if both numbers are even or odd  or the left is even and the right is odd
        change them only if the number in the left is bigger than the one in the right */
        for( i = 0 ; i <= 6; i++ ) {
            if( is_even(array[i]) && is_even(array[i+1])) {
                if( array[i] > array[i+1]) {
```

```
                    swap       = array[i];
                    array[i]   = array[i+1];
                    array[i+1] = swap;
                }
            }
            if( !is_even(array[i]) && !is_even(array[i+1])) {
                if( array[i] > array[i+1]) {
                    swap       = array[i];
                    array[i]   = array[i+1];
                    array[i+1] = swap;
                }
            }
            if( is_even(array[i]) && !is_even(array[i+1])) {
                swap       = array[i];
                array[i]   = array[i+1];
                array[i+1] = swap;
            }
        }
        i = 0;
    }

    printf("\nAfter we sort them they are: ");
    for ( i = 0 ; i <= 7; i++) {
        printf(" %d ", array[i]);
    }
    return 0;
}

bool is_even(int value) {
    if( value % 2 == 0 )
        return true;
    return false;
}

int is_sorted(int a[]) {

    int i;

    for(i = 0; i <= 6; i++) {
    /* the numbers aren't sorted  even if there is one left number that is even and the
     right is odd */
        if( is_even(a[i]) && !is_even(a[i+1]) ) {
            return 0;
            break;
        }
        /* if there is a left number that is bigger than the right it's ok as long as the left is
        odd and the right is even, in any other case the numbers aren't sorted */
        if(a[i] > a[i+1]) {
            if(!is_even(a[i]) && is_even(a[i+1])) {
                continue;
            }
            else {
```

```
                return 0;
                break;
            }
        }
    }
    return  1;
}
```

Αποφεύγοντας τα πράγματα που είπαμε από πάνω ο κώδικας γράφεται ως εξής:

```
#include <stdio.h>
#include <stdbool.h>

bool is_even(int value);
int is_sorted(int a[]);

int main()
{
    /* Initialization of swap */
    int array[8], i , swap = 0;

    for ( i = 0 ; i <= 7; i++){
        scanf("%d", &array[i]);
    }

    printf("The integers you put are:");
    for ( i = 0 ; i <= 7; i++){
        printf("  %d", array[i]);
    }
    /* Instead of  is_sorted  == false better to write !is_sorted */
    while( !is_sorted(array)) {
        for( i = 0 ; i <= 6; i++ ) {
            if( is_even(array[i]) && is_even(array[i+1])) {
                if( array[i] > array[i+1]) {
                    swap       = array[i];
                    array[i]   = array[i+1];
                    array[i+1] = swap;
                }
            }
            if( !is_even(array[i]) && !is_even(array[i+1])) {
                if( array[i] > array[i+1]) {
                    swap       = array[i];
                    array[i]   = array[i+1];
                    array[i+1] = swap;
                }
            }
            if( is_even(array[i]) && !is_even(array[i+1])) {
                swap       = array[i];
                array[i]   = array[i+1];
                array[i+1] = swap;
            }
        }
        i = 0;
    }
```

```
    printf("\nAfter we sort them they are: ");
    for ( i = 0 ; i <= 7;  i++ ) {
        printf(" %d ", array[i]);
    }

    return 0;
}

/* It's easier to see what the function will return this way (with the extra else),also its better not to return true or false from a function*/
bool is_even(int value) {
    if( value % 2 == 0 )
        return 1;
    else {
        return 0;
    }
}

int is_sorted(int a[]){

    int i;

    for(i = 0; i <= 6; i++) {
        if(is_even(a[i]) && !is_even(a[i+1])) {;
            return 0;
            break;
        }
        if(a[i] > a[i+1]) {
            if(!is_even(a[i]) && is_even(a[i+1])) {
                continue;
            }
            else {
                return 0;
                break;
            }
        }
    }
    return  1;
}
```

3. Προσοχή στις επαναληπτικές δομές καθώς είναι σημείο που συχνά βρίσκεις σφάλματα (bugs). Η επανάληψη μπορεί να είναι εύκολη, το πότε να σταματήσουμε όμως είναι δύσκολο. Ακολουθήστε τους θεμελιώδης κανόνες του σχεδιασμού:

- σκεφτείτε τι θέλετε να είναι αληθές μετά την επανάληψη, αυτό διαμορφώνει την συνθήκη τερματισμού.

Για παράδειγμα στον κώδικα μας η is_sorted θα εκτελείται μέχρι να φτάσουμε στο τέλος του πίνακα ή αν πετύχουμε πρώτα ζυγό και μετά μονό αριθμό ή αν πετύχουμε αριθμό που είναι σε πιο αριστερή θέση στον πίνακα και είναι μεγαλύτερος από τον επόμενο αριθμό εκτός αν ο αριστερός αριθμός είναι μονός και ο δεξιός αριθμός είναι ζυγός.

- η άρνηση της συνθήκης τερματισμού είναι η συνθήκη επανάληψης.

Παράδειγμα το **(**a**[**i**] >** a**[**i**+**1**])** είναι ισοδύναμο με το a**[**i**] <=** a**[**i**+**1**]**
μετά απλώς συμπληρώνουμε το σώμα της επανάληψης [26].

4. Μην προσπαθείτε να κρύψετε έναν δείκτη σε μια typedef
   - αν είναι δείκτης κάντε το έτσι ώστε να φαίνεται ότι είναι δείκτης
   - η αφαίρεση (abstraction) είναι για κρύβουμε τις ασήμαντες λεπτομέρειες

Κακό παράδειγμα typedef

```
typedef struct date * date;
bool date_equal(const date lhs, const date rhs);
```

τι είναι const;

```
bool date_equal(const struct date * lhs,
      const struct date * rhs);

bool date_equal( struct date * const lhs,
       struct date * const  rhs );
```

Για αυτές τις 2 εκδοχές είναι const ο δείκτης ή μήπως είναι η τιμή του δείκτη;

Εναλλακτικά(καλές χρήσεις typedef) :

date.h

```
   typedef struct date date;
   bool date_equal( const date * lhs,   const date * rhs );
```

ή

date.h

```
struct date;
bool date_equal(const struct date * lhs,const struct date * rhs);
```

όπου const είναι όλο το αντικείμενο [27].

5. Η typedef δεν δημιουργεί νέο τύπο. Έστω το παράδειγμα:

```
typedef int mile;
typedef int kilometer;

void weak(mile lhs, kilometer rhs) {
     lhs = rhs;
}
```

Έτσι ενώ η ανάθεση που κάνουμε είναι σωστή συντακτικά, βάση της λογικής δεν είναι σωστή γιατί εξισώνει διαφορετικά πράγματα.

6. Σκεφτείτε να χρησιμοποιήσετε έναν τύπο περιτυλίγματος στην θέση του.

```
typedef struct { int value; } mile;
typedef struct { int value; } kilometer;

void strong(mile lhs, kilometer rhs) {
     lhs = rhs;
}
```

Εδώ η λογική είναι σωστή αλλά η ανάθεση δεν είναι σωστή συντακτικά. Θα έπρεπε να γραφτεί σαν: lhs.value = rhs.value; [28].

7. Οι απαριθμήσεις (enums) είναι ένας βολικός τρόπος σύνδεσης σταθερών ακέραιων τιμών με ονόματα. Τα enums είναι ασθενής τύποι και ένας enum απογραφέας είναι τύπου int και όχι τύπου enum [29]. Παράδειγμα:

```
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,JUL, AUG, SEP, OCT,
    NOV, DEC};
```

Εδώ, οι σταθερές για τους μήνες στην απαρίθμηση months αντιστοιχούν στους αριθμούς 1 εώς 12.

8. Αν αλλάξεις ένα .h αρχείο πρέπει να ξανακάνεις μεταγλώττιση (compile) όλα τα αρχεία που περιλαμβάνουν(#include) το .h αρχείο [30].

9. Για να γίνει η δήλωση ενός αφαιρετικού τύπου δεδομένων ή ΑΤΔ (Abstraction data type ή ADT) θα πρέπει να:

- Βάλουμε στο .h αρχείο τα ελάχιστα #include που χρειάζονται.
- Να γίνει το **typedef**.
- Όλες οι χρήσεις αυτού του τύπου πρέπει να είναι δείκτες.

Παράδειγμα:

wibble.h

```
/* minimum includes*/
...
/* wibble hasn't defined yet */
typedef struct wibble_tag wibble;

/* all uses of wibble has to be pointers */
wibble * wopen( const char * filename);

int wclose(wibble * stream); [31].
```

10. Στους αφαιρετικούς τύπους δεδομένων υπάρχει αδιαφανής τύπος διαχείρισης μνήμης, δηλαδή οι πελάτες δεν μπορούν να δημιουργήσουν αντικείμενα, δεδομένου ότι δεν ξέρουν πόσα bytes που καταλαμβάνουν, παράδειγμα:

Αν έχουμε ένα αρχείο wibble.h με

```
...
typedef struct wibble wibble;
...
```

Και

```
#include "wibble.h"


void client(void) {
wibble * pointer;              /* This is allowed*/
...
wibble value;                  /*This is not allowed*/
...
ptr = malloc( sizeof (*ptr));  /*This is not allowed*/
} [32].
```

11. Ένας αφαιρετικός τύπος δεδομένων μπορεί να διακηρύξει το μέγεθός του.

Εάν το γράψουμε όπως στο παρακάτω παράδειγμα μπορούμε να πετύχουμε τα εξής :

- Οι πελάτες μπορούν τώρα να εκχωρήσουν(allocate) τη μνήμη.
- Η πραγματική εικόνα παραμένει αφαιρετική.

wibble.h

```
typedef struct wibble {
    /* declare size of ADT */
    unsigned char size[16];
} wibble;

bool wopen( wibble *,  const char *);
void wclose( wibble * );
```

```
#include "wibble.h"

void client( const char * name) {
    /* the implementation of the ADT remains hidden */
    wibble w;

    if (wopen( &w, name ))
    ...
    wclose( &w );
}
```

[33].

4. ΔΟΜΗΜΕΝΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

4.1 Εισαγωγή

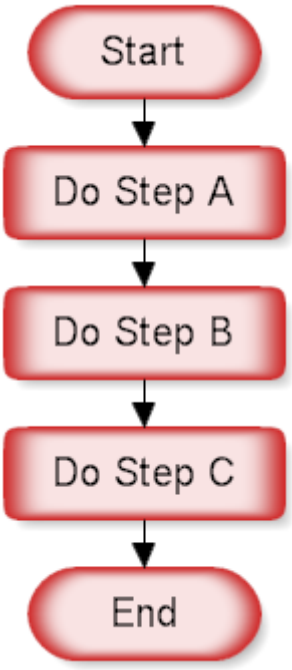
Σε αυτό το κεφάλαιο θα εισάγουμε την έννοια του δομημένου προγραμματισμού. Πρωταρχικός σκοπός του δομημένου προγραμματισμού είναι να δημιουργεί ευανάγνωστα και κατανοητά προγράμματα υψηλού βαθμού.

4.2 Δομημένος Προγραμματισμός

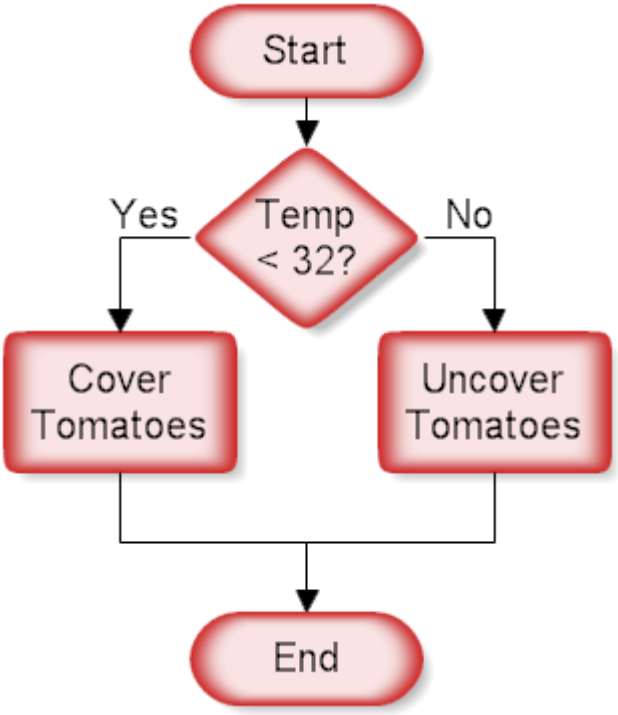
Ο δομημένος προγραμματισμός είναι συνυφασμένος με την κατασκευή προγραμμάτων που περιέχουν μόνο, όσο αυτό είναι εφικτό, τους εξής τρεις τύπους εντολών:

1. Ακολουθιακές (απόδοση τιμής, εισόδου/εξόδου (input/output ή I/O), σύνθετη εντολή, ανάκληση συναρτήσεων).
2. Υποθετικές (**if**, **else, switch**).
3. Επαναληπτικές (**while, do while, for)**

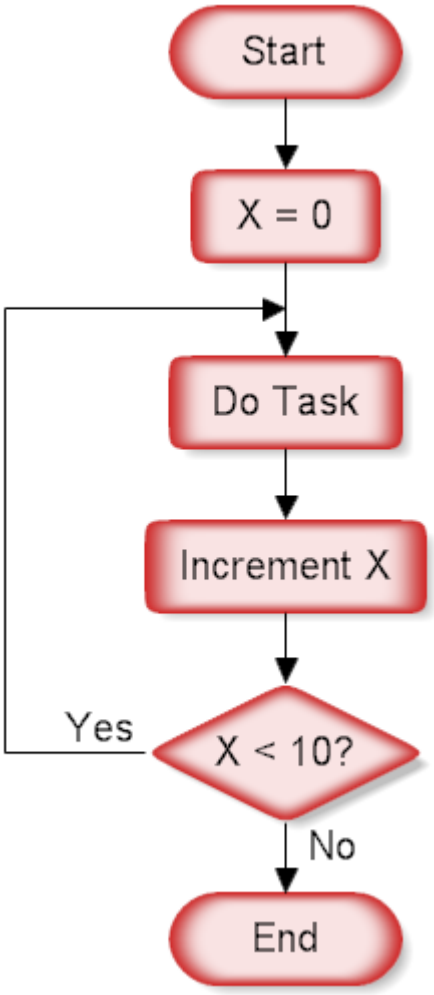
Οι ακολουθιακές εντολές εκτελούν μία λειτουργία και μετά συνεχίζουν στην επόμενη. Οι υποθετικές εντολές ελέγχουν, ανάλογα με την τιμή μίας ή πολλών μεταβλητών, ποια εντολή πρέπει να εκτελεστεί μετά. Το μοντέλο μίας υπό συνθήκη εντολής σαν διάγραμμα ροής φαίνεται στο σχήμα 2. Τέλος μια επαναληπτική εντολή εκτελεί συνεχώς ένα σύνολο εντολών μέχρι να ικανοποιηθεί μία συνθήκη (σχήμα 3). Ένα καλά δομημένο πρόγραμμα περιέχει μόνο αυτά τα τρία είδη εντολών**.**



Σχήμα 1: Ακολουθιακή εντολή.



Σχήμα 2: Υπό συνθήκη εντολή.



Σχήμα 3: Επαναληπτική εντολή.

Εντολές όπως η **goto** εξαιρούνται πάντα. Το πιο κύριο χαρακτηριστικό ενός δομημένου κώδικα είναι ότι αποτελείται από μέρη προγράμματος το ένα μέσα στο άλλο (nested) που έχουν ένα σημείο εισόδου και ένα σημείο εξόδου (single entry single exit). Εξάλλου είναι μαθηματικά αποδεδειγμένο πως ένα πρόγραμμα μπορεί να αποτελείται μόνο από τις εντολές **if/else, while/do, while/for** και άλλες κατάλληλες ακολουθιακού τύπου (Θεώρημα C. Boehm και G. Jacopini 1966).

4.3 Εντολές if-else

Η σύνταξη της εντολής if-else είναι:

```
if(παράσταση) {
    μπλοκ εντολών 1
}
else {
    μπλοκ εντολών 2
}
```

Αν η (παράσταση) είναι αληθής (έχει τιμή διάφορη από το μηδέν), θα εκτελεσθεί το μπλοκ εντολών 1, αλλιώς θα εκτελεσθεί το μπλοκ εντολών 2. Επειδή η **if** απλώς ελέγχει την αριθμητική τιμή μιας παράστασης, μπορούν να γίνουν ορισμένες συντομεύσεις στον κώδικα. Ο προφανής είναι να γράψουμε:

```
if(παράσταση)        αντί για        if(παράσταση != 0)
```

Το τμήμα από το **else** και μετά είναι προαιρετικό. Αν δεν υπάρχει και η (παράσταση) είναι ψευδής, ο έλεγχος θα πάει στην επόμενη εντολή του προγράμματος, δηλαδή στην πρώτη εντολή μετά το **if**.

Η προαιρετικότητα του **else** μπορεί να προκαλέσει προβλήματα στην αναγνωσιμότητα προγραμμάτων με εμφωλευμένες εντολές **if**, αν δεν υπάρχει σαφής ομαδοποίηση με άγκιστρα. Αν δεν υπάρχουν άγκιστρα που να επιβάλλουν συγκεκριμένη δομή, κάθε **else** αντιστοιχεί στην αμέσως προηγούμενη **if** που δεν έχει **else**.

Προσοχή! Ο μεταγλωττιστής δεν ασχολείται με τη στοίχιση. Παράδειγμα:

```
if (n > 0) {
    if (a > b)
        z = a;
    else
        z = b;
}
```

Αν θέλαμε το else να αντιστοιχεί στο πρώτο **if**, έπρεπε να το γράψουμε έτσι:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

Μία συνηθισμένη χρήση της εντολής if είναι όταν θέλουμε να ελέγξουμε μία σειρά από επάλληλες συνθήκες, και ανάλογα με το ποια θα βρεθεί ότι ισχύει πρώτη, να εκτελέσουμε ένα μπλοκ εντολών. Αυτό γίνεται ως εξής:

```
if (παράσταση1)
    μπλοκ εντολών 1
else if (παράσταση 2)
```

```
    μπλοκ εντολών 2
.....
else if (παράσταση n-1)
    μπλοκ εντολών n-1

else
    μπλοκ εντολών n
```

Η προηγούμενη εντολή if θα μπορούσε να γραφεί σε μία εκδοχή με στοίχιση ως εξής:

```
if (παράσταση1)
    μπλοκ εντολών1
else
    if (παράσταση2)
        μπλοκ εντολών 2
  .....
        else
            if (παράσταση n-1)
                μπλοκ εντολών n-1
            else
                μπλοκ εντολών n
```

Η συγκεκριμένη στοίχιση, όμως, μάλλον κάνει πιο δυσανάγνωστη την εντολή, οπότε είναι καλό να μην την ακολουθήσει κάποιος.
Ένα καλό παράδειγμα είναι το:

```
if (x > 0) {
    sign = 1;
    printf("Number is positive \n");
}
else if (x < 0) {
    sign = -1;
    printf("Number is negative \n");
}
else {
     sign = 0;
     printf("Number is zero \n");
} [34].
```

4.4 Εντολή switch

Η σύνταξη της εντολής **switch** είναι:

```
switch(παράσταση) {
    case σταθερά 1:
    εντολές 1
    case σταθερά 2:
    εντολές 2
……
    default:
        εντολές
 }
```

- Η **switch** είναι μία διακλαδωμένη εντολή απόφασης που λειτουργεί όπως περιγράφεται στη συνέχεια.
- Η (παράσταση) πρέπει να είναι ακέραια. Υπολογίζεται η τιμή της.

- Κάθε (σταθερά) i πρέπει να είναι ακέραια σταθερά και δεν πρέπει να υπάρχουν δύο **case** με την ίδια σταθερά.
- Αν η τιμή που έχει η (παράσταση) ισούται με κάποια (σταθερά) i, τότε ο έλεγχος μεταφέρεται στις (εντολές) i. Αν όχι, ο έλεγχος μεταφέρεται στις (εντολές) (μετά το **default**).
- Οι εντολές i(1,2…) και εντολές στην **default** δεν είναι απαραίτητο να συνιστούν μπλοκ εντολών, δηλαδή να είναι κλεισμένες μέσα σε { και }.

Μετά την εκτέλεση των εντολών σε μία **case**, ο έλεγχος μεταφέρεται στις εντολές της επόμενης **case**, εκτός αν τελευταία εντολή της προηγούμενης **case** είναι η **break**. Δεν είναι καλή πρακτική οι εντολές μίας **case** να συνεχίζουν με αυτές της επόμενης, εκτός αν ισχύει το επόμενο.

- Είναι δυνατόν για περισσότερες τιμές της μίας για την (παράσταση) να θέλουμε να εκτελεσθεί η ίδια ομάδα εντολών. Σ΄ αυτήν την περίπτωση, τοποθετούμε τα αντίστοιχα **case** συνεχόμενα και βάζουμε την ομάδα εντολών στο τελευταίο από αυτά.
- Η περίπτωση **default** είναι προαιρετική. Αν δεν υπάρχει και δεν ταιριάζει καμία **case**, ο έλεγχος μεταφέρεται μετά την εντολή **switch**.

Παράδειγμα:

```
switch (x) {
    case 1:
        printf("one\n");
        break;
    case 2:
    case 3:
        printf("two or three\n");
        break;
    case 4:
        printf("four\n");
        break;
    default:
        printf("other\n");
        break;
}
```
[35].

4.5 Εντολές βρόχου while

Μία πιθανή σύνταξη της εντολής **while** είναι:

```
while( παράσταση ) {
    μπλοκ εντολών
}
```

Αρχικά υπολογίζεται η παράσταση. Αν έχει τιμή διάφορη του μηδενός (είναι αληθής) εκτελείται το μπλοκ εντολών. Υπολογίζεται πάλι η παράσταση και ενόσω είναι αληθής, θα γίνεται ο κύκλος εκτέλεσης του μπλοκ εντολών, μέχρι να γίνει η παράσταση ψευδής (δηλαδή ίση με το μηδέν). Παράδειγμα:

```
while(( c = getchar() != EOF )) {
    if(c = ' ' ) {
        whitespaces++;
    }
}
```

Άλλη πιθανή σύνταξη της εντολής while είναι:

```
do {
```

```
    (μπλοκ εντολών)
}
while (παράσταση)
```

Αρχικά εκτελείται το μπλοκ εντολών. Μετά, υπολογίζεται η παράσταση. Αν έχει τιμή διάφορη του μηδενός (είναι αληθής) εκτελούνται πάλι το μπλοκ εντολών και συνεχίζεται ο κύκλος εκτέλεσής του, μέχρι να γίνει η παράσταση ψευδής (δηλαδή ίση με το μηδέν).

Για λόγους καλύτερης αναγνωσιμότητας του προγράμματος, ακόμα και αν μπλοκ εντολών αποτελείται από μία μόνο εντολή, συνήθως τη βάζουμε μέσα σε { και }. Παράδειγμα:

```
i=1;

do {
    i = i + 2;
} while( i < 10 );
```

Η βασική διαφορά των δύο εκδοχών της εντολής **while** είναι ότι στην πρώτη περίπτωση το μπλοκ εντολών μπορεί και να μην εκτελεσθεί καμία φορά, αν η παράσταση είναι αρχικά ψευδής, ενώ στην εκδοχή **do** / **while**, το μπλοκ εντολών θα εκτελεσθεί τουλάχιστον μία φορά, αφού η παράσταση ελέγχεται στο τέλος [36].

4.6 Εντολές βρόχου for

Η σύνταξη της εντολής for είναι:

```
for ( (παράσταση 1)  ; (παράσταση 2)  ; (παράσταση 3) ) {
    μπλοκ εντολών
}
```

Διαδικαστικά, ισοδυναμεί με τις εξής εντολές:

```
παράσταση 1;
while (παράσταση 2) {
    μπλοκ εντολών
    (παράσταση 3);
}
```

Δηλαδή, αρχικά υπολογίζεται η παράσταση 1. Συνήθως, πρόκειται για την αρχικοποίηση ενός δείκτη που ελέγχει μία επαναληπτική διαδικασία. Στη συνέχεια, εκτελείται επαναλαμβανόμενα το μπλοκ εντολών, που αποτελεί το σώμα της διαδικασίας, ενόσω η παράσταση 2, που συνήθως ελέγχει την τιμή του δείκτη ελέγχου, είναι αληθής. Πριν τη διεξαγωγή νέας επανάληψης, εκτελείται και η παράσταση 3, που συνήθως μεταβάλλει τον δείκτη ελέγχου.

```
x = 0;

for( i = 1; i < 11; i++) {
    /*sum from 1 till 10*/
    x = x + i;
}
```

Είναι δυνατόν κάποια από τις παράσταση 1, 2 ή 3 να μην υπάρχει, συνήθως η πρώτη ή/και η τρίτη. Αν δεν υπάρχει η δεύτερη, τότε δεν γίνεται έλεγχος για τερματισμό του βρόχου, οπότε πρέπει αυτό να γίνει βιαίως με κάποιο τρόπο μέσα από το σώμα της

επανάληψης. Σε κάθε περίπτωση, όποια παράσταση i και να λείπει, τα ; υπάρχουν κανονικά.

Η εντολή

```
for ( ; ; )
    μπλοκ εντολών
```

είναι η κλασική υλοποίηση ενός ατέρμονος βρόχου.

Επίσης, στην C υπάρχει και ο τελεστής , (κόμμα), με τον οποίο μπορούμε να κατασκευάσουμε μία σύνθετη παράσταση που θα υπολογισθεί από αριστερά προς τα δεξιά. Η τιμή της σύνθετης παράστασης ισούται με την τιμή της δεξιότερης από τις απλές που την συνιστούν.

Μερικές φορές, θέλουμε να γράψουμε μία εντολή for που θα ελέγχεται από περισσότερους του ενός δείκτες. Αυτό γίνεται χρησιμοποιώντας τον τελεστή , στην (παράσταση 1) και στην (παράσταση 3). Παράδειγμα:

```
for ( i = 0, j = n ; i <= j ; i++, j--) {
    printf("%d \n", i * j);
} [37].
```

4.7 Εντολές break και continue

Μία χρήση της εντολής **break** είναι αυτή που είδαμε στην εντολή **switch**, για τον τερματισμό της ακολουθίας εντολών που θα εκτελεσθούν όταν ταιριάξει κάποια συγκεκριμένη **case**.

Επίσης, με την εντολή **break**, μπορούμε να τερματίσουμε βιαίως τις επαναλήψεις ενός βρόχου (**while**, **do/while** ή **for**), πριν ισχύσει η συνθήκη τερματισμού (ή όταν ο βρόχος είναι ατέρμων). Παράδειγμα:

```
while (!finished) {
    ……….
    if (bad_data)
        break;
    ……….
}
```

Με την εντολή **continue**, ο έλεγχος στο εσωτερικό ενός βρόχου μεταφέρεται στο τέλος του, για να αρχίσει η επόμενη επανάληψη. Παράδειγμα:

```
for ( i = 0 ; i < n ; i++) {
    /* Ignore negative elements */
    if (a[i] < 0)
        continue;
   ... /* Process positive elements */
} [38].
```

4.8 Εντολή goto

Όπως κάθε γλώσσα προγραμματισμού, έτσι και η C, έχει μία εντολή για ρητή μεταφορά του ελέγχου σε κάποιο άλλο σημείο (ετικέτα) του προγράμματος. Στην C, η εντολή αυτή είναι η **goto**. Η **goto** μπορεί να χρησιμοποιηθεί σε πολύ εξειδικευμένες περιπτώσεις, για παράδειγμα όταν θέλουμε να γίνει βίαιη έξοδος από εμφωλευμένους βρόχους, όταν ο έλεγχος βρίσκεται σε κάποιο εσωτερικό βρόχο. Αυτό δεν μπορεί να γίνει με την εντολή **break**. Παράδειγμα:

```
for (i=0 ; i < n ; i++)
    for (j=0 ; j < m ; j++)
        if (a[i] == b[j])
            goto found;
    ..... /* Didn't find common element */
    found:
    ..... /* Found a[i] == b[j] */
```

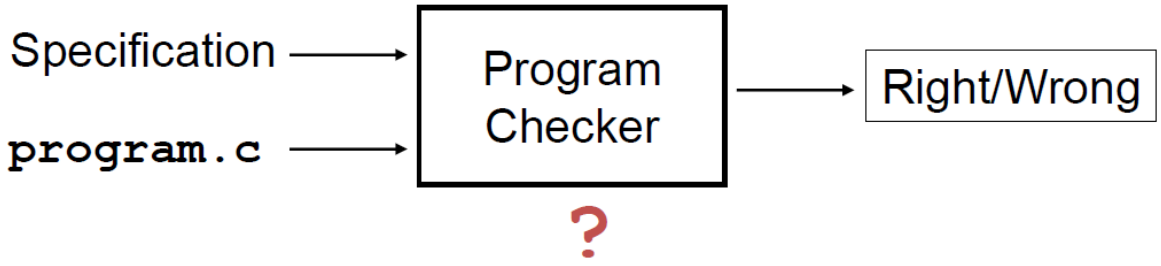
Η εντολή **goto** βλάπτει σοβαρά την ιδέα του δομημένου προγραμματισμού, γιατί μπορεί να οδηγήσει σε προγράμματα δυσανάγνωστα και δύσκολα συντηρήσιμα, και για αυτό πρέπει να χρησιμοποιείται σπάνια, αν όχι καθόλου. Άλλωστε είναι μαθηματικά αποδεδειγμένο ότι σε κάθε περίπτωση η **goto** μπορεί να γραφτεί ισοδύναμα με εντολές **if** / **else**, **for**, **while** [39].

5. TESTING - DEBUGGING

Ο λειτουργικός έλεγχος (testing) και η αποσφαλμάτωση (debugging) συχνά αναφέρονται ως μια φράση αλλά δεν αντιστοιχούν στην ίδια διαδικασία. Για χάριν απλούστευσης, η αποσφαλμάτωση είναι διαδικασία που ακολουθείς όταν το πρόγραμμα δεν λειτουργεί. Ενώ ο λειτουργικός έλεγχος είναι μια συστηματική προσπάθεια να χαλάσεις ένα πρόγραμμα που πιστεύεις ότι λειτουργεί.

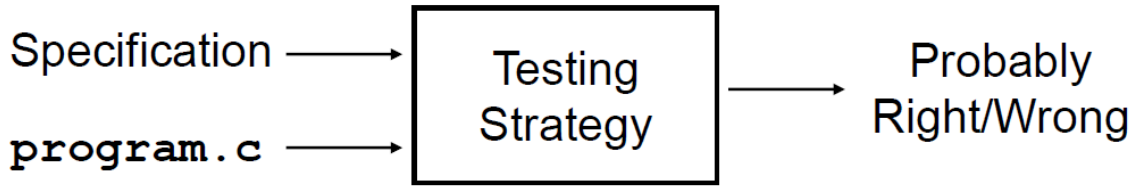
Ο Edsger Dijkstra έκανε την περίφημη παρατήρηση ότι ο έλεγχος μπορεί να αποδείξει την παρουσία σφαλμάτων, αλλά όχι την απουσία τους. Η ελπίδα του ήταν ότι τα προγράμματα μπορούν να γίνουν σωστά κατά την κατασκευή τους, έτσι ώστε να μην υπάρχουν λάθη και έτσι να μην υπάρχει ανάγκη για έλεγχο. Αν και αυτός είναι ένα καλός στόχος, δεν είναι ρεαλιστικός για σημαντικά προγράμματα [40].

Ιδανικά θα θέλαμε να αποδείξουμε ότι το πρόγραμμα είναι σωστό. Αλλά μπορούμε να αποδείξουμε τις ιδιότητες του προγράμματος ή ότι έστω τερματίζει;



Σχήμα 4 : Παράδειγμα λογικής ελέγχου προγράμματος [41].

Στην πραγματικότητα πρέπει να πείσουμε τους εαυτούς μας ότι το πρόγραμμα μας δουλεύει.



Σχήμα 5 : Παράδειγμα στρατηγικής ελέγχου [42].

Το να σκεφτόμαστε για πιθανά προβλήματα καθώς προγραμματίζουμε είναι μια καλή αρχή. Ο συστηματικός έλεγχος, από εύκολες δοκιμές ως τις πιο πολύπλοκες, βοηθά στο να διασφαλιστεί ότι τα προγράμματα αρχίζουν να λειτουργούν σωστά και να παραμείνουν σωστά καθώς μεγαλώνουν. Ο αυτοματισμός βοηθά στην εξάλειψη των χειροκίνητων διαδικασιών και ενθαρρύνει τις εκτεταμένες δοκιμές.

Ένας τρόπος για να γράψει κάποιος κώδικα χωρίς σφάλματα (bug free code) είναι να τον δημιουργήσει με ένα πρόγραμμα. Όταν μια προγραμματιστική λειτουργία είναι τόσο γνωστή σε σας ώστε να γράφεται τον κώδικα μηχανικά, τότε θα πρέπει να αυτοματοποιηθεί. Μια κοινή περίπτωση προκύπτει όταν ένα πρόγραμμα μπορεί να παραχθεί από μια προδιαγραφή σε κάποια εξειδικευμένη γλώσσα. Για παράδειγμα, μεταγλωττίζουμε κώδικα υψηλού επιπέδου γλώσσας σε κώδικα assembly,

χρησιμοποιούμε κανονικές εκφράσεις (regular expressions) για να καθορίσουμε μοτίβα σε ένα κείμενο, χρησιμοποιούμε συμβολισμούς όπως SUM(A1:A50) για να αναπαραστήσουμε λειτουργίες σε μια περιοχή κελιών σε ένα υπολογιστικό φύλλο. Σε τέτοιες περιπτώσεις, αν ο μεταγλωττιστής είναι σωστός και αν ακολουθούνται οι προδιαγραφές, τότε και το πρόγραμμα που προκύπτει θα είναι επίσης σωστό.

Είναι αυτονόητο ότι όσο νωρίτερα βρεθεί ένα πρόβλημά στο πρόγραμμα τόσο καλύτερα. Αν σκέφτεστε μεθοδικά για τον κώδικα σας την ώρα που τον γράφεται, τότε, μπορείτε παράλληλα να επαληθεύεται τα απλά χαρακτηριστικά του προγράμματος. Αυτό έχει σαν αποτέλεσμα να ολοκληρώσετε το πρώτο στάδιο ελέγχου πριν ακόμα ολοκληρωθεί το ίδιο το πρόγραμμα και να εξαλείψετε συγκεκριμένου τύπου προβλήματα.

5.1 Εξωτερικός έλεγχος (External testing)

Η κατεύθυνση που ακολουθούμε εδώ είναι να σχεδιάζουμε τα δεδομένα με τα οποία θα ελέγξουμε το πρόγραμμα καθώς το γράφουμε. Ακολουθούν μερικές τεχνικές που μας βοηθούν σε αυτό το στόχο.

5.1.1 Έλεγχος των οριακών περιπτώσεων (boundaries)

Είναι πολύ σημαντικό να χρησιμοποιήσετε ελέγχους που καλύπτουν σε βάθος, όλες τις πιθανές «συμπεριφορές» μίας συνάρτησης. Σε βαθμό που είναι εφικτό, θα πρέπει να ελέγξετε όλα τα πιθανά μονοπάτια που υπάρχουν μέσα στον κώδικα. Επίσης είναι σημαντικό να ελεγχθούν και οι οριακές περιπτώσεις (boundaries). Αν ένα τμήμα του κώδικά πρόκειται να αποτύχει τότε είναι πιο πιθανό να το κάνει σε μια οριακή περίπτωση. Με την ίδια λογική αν είναι πρόγραμμα λειτουργεί σωστά στις οριακές περιπτώσεις τότε είναι πολύ πιθανό ότι λειτουργεί σωστά και στις υπόλοιπες.

Είναι σημαντικό να αναπτύξουμε την ικανότητά μας, να φανταζόμαστε όλες τις δυνατές και τις πιθανές περιπτώσεις (ιδίως τις πιο σπάνιες). Μερικές φορές το παραπάνω είναι πολύ δύσκολο, όπως για παράδειγμα όταν προσπαθούμε να ελέγξουμε πότε μία διεπαφή τερματίζει απρόσμενα στην περίπτωση που χρησιμοποιείται από χρήστες με εσφαλμένο τρόπο. Σε αυτές τις περιπτώσεις, ίσως να ήταν καλύτερο να ζητήσετε από κάποιους μη εξοικειωμένους χρήστες να «παίξουν» με την εφαρμογή που φτιάξατε (μιας και οι σχεδιαστές ενός συστήματος γνωρίζουν πως πρέπει να λειτουργεί το σύστημά τους, τους είναι δύσκολο να κάνουν μη σωστή χρήση του).

Για παράδειγμα, το παρακάτω κομμάτι κώδικά λειτουργεί περίπου όπως η fgets και διαβάζει χαρακτήρες μέχρι να διαβάζει τον χαρακτήρα αλλαγής γραμμής (newline) ή να γεμίσει την προσωρινή μνήμη (buffer).

```
int i;
char s[MAX];

for (i = 0; (s[i] = getchar()) != '\n' || i < MAX-1; ++i)  {
      s[--i] = '\0';
}
```

Ο πρώτος έλεγχος οριακών τιμών είναι εύκολος είναι να ελέγξουμε αν είναι η κενή γραμμή. Αν ξεκινήσετε με μια γραμμή που περιέχει μόνο το χαρακτήρα (newline) είναι εύκολο να δείτε ότι η επαναληπτική διαδικασία σταματά στο πρώτο βήμα. Το i θα έχει τιμή 0 οπότε η τελευταία εντολή θα πάει να γράψει ένα χαρακτήρα τερματισμού '\0' στη θέση s[-1]. Έτσι με αυτό τον έλεγχο ανακαλύψαμε το πρώτο μας λάθος

Ξαναγράφουμε την επαναληπτική διαδικασία μας με πιο συμβατικό τρόπο

```
for (i = 0; i < MAX-1; i++) {
    if ((s[i] = getchar()) == '\n') {
        break;
    }
}
s[i] = '\0';
```

Επαναλαμβάνοντας ξανά το αρχικό τεστ είναι εύκολο να επαληθεύουμε ότι όλα θα δουλέψουν κανονικά. Η επανάληψη θα σταματήσει στο πρώτο βήμα με i = 0 και στη θέση 0 του πίνακα s θα αποθηκεύσουμε ένα τερματικό χαρακτήρα. Υπάρχουν όμως και άλλες οριακές περιπτώσεις που πρέπει να ελέγξουμε. Αν για παράδειγμα η είσοδο μας περιέχει πολύ μεγάλες γραμμές ή καθόλου χαρακτήρες αλλαγής γραμμής (newline). Και σε αυτή την περίπτωση όμως μπορούμε να επαληθεύσουμε ότι το πρόγραμμα θα λειτουργήσει κανονικά εφόσον χρησιμοποιούμε τον έλεγχο για το πλήθος των αναγνωσμένων χαρακτήρων MAX-1. Τι θα γίνει όμως στην περίπτωση που η getchar() επιστρέψει αμέσως τον χαρακτήρα EOF (end of file) ;

Σε αυτήν την περίπτωση θα τροποποιήσουμε τον κώδικα μας για να προσθέσουμε τον επιπλέον έλεγχο:

```
for ( i=0; i<MAX-1; i++) {
    if ( (s[i] = getchar()) == '\n' || s[i] == EOF ) {
        break;
    }
}
s[i] = '\0';
```

Ο έλεγχος οριακών συνθηκών λοιπόν δείχνει αποτελεσματικός για την εύρεση λαθών όπου τα όρια μας ξεφεύγουν κατά μία μονάδα. Με συχνή εξάσκηση αυτός ο τρόπος μπορεί να γίνει τόσο οικείος σε σας όποτε και όλα τα ασήμαντα λάθη θα εξαλείφονται πριν ακόμα συμβούν.

5.1.2 Statement testing (έλεγχος εντολών)

Ο έλεγχος εντολών είναι μια τεχνική που προϋποθέτει την εκτέλεση όλων των εντολών του πηγαίου κώδικα τουλάχιστον μια φορά. Χρησιμοποιείται σαν κριτήριο καταμέτρησης όλων των εντολών που εκτελούνται στο πρόγραμμα. Χρησιμοποιώντας αυτή την τεχνική μπορούμε να ελέγξουμε τι είναι επιτρεπτό να κάνει το πρόγραμμά μας και τι όχι. Μπορούμε να ελέγξουμε επίσης την ποιότητα του κώδικα και τις ροές από τα διαφορετικά μονοπάτια. Το μόνο μειονέκτημα αυτής τις τεχνικής είναι ότι δεν μπορούμε να ελέγξουμε για εσφαλμένες εντολές.

Ακολουθάει το παρακάτω απλοϊκό παράδειγμα σε ψευδοκώδικα:

```
Read A
Read B
if A>B
    Print “A is greater than B”
else
    Print "B is greater than A"
endif
```

1η Περίπτωση : **A = 5, B = 2**
Αριθμός των εντολών που εκτελέστηκαν : **5**

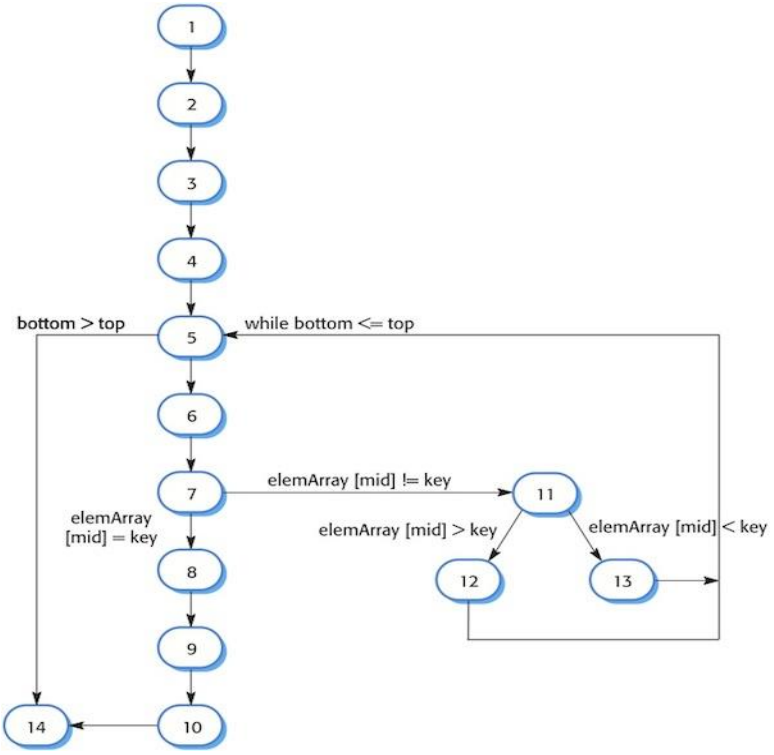
Συνολικός αριθμός εντολών :**7**
Ποσοστό πληρότητας : **5/7 = 71%**

2η Περίπτωση : **A =2, B = 5**
Αριθμός των εντολών που εκτελέστηκαν : **6**
Συνολικός αριθμός εντολών : **7**
Ποσοστό πληρότητας : **6/7 = 85.2%**

5.1.3 Path testing (έλεγχος πιθανών ροών εκτέλεσης)

Μια διαφορετική τεχνική ελέγχου του προγράμματος είναι ο έλεγχος πιθανών ροών εκτέλεσης όπου φροντίζεις κάθε διαφορετική ροή του προγράμματος να εκτελεστεί τουλάχιστον μια φορά. Συνήθως αυτό γίνεται με κάποιο αυτοματοποιημένο εργαλείο ελέγχου. Το να ελέγχεις όλα τα πιθανά μονοπάτια όμως δεν σημαίνει ότι θα εντοπίσεις και όλα τα σφάλματα στον κώδικά. Μερικά σφάλματα για παράδειγμα προκύπτουν όταν οι προγραμματιστές ξεχάσουν να προσθέσουν κάποιες εντολές στον κώδικα, οπότε και δεν θα υπάρξει μονοπάτι που θα μπορέσουν να ελεγχθούν. Εδώ σημαντικό ρόλο παίζει και η σειρά με την οποία θα ελεγχθεί το κάθε μονοπάτι. Για παράδειγμα έχουμε αν ένα πρόγραμμα που αποτελείται από 3 τμήματα α,β,γ και ελέγχοντας το μονοπάτι που τα εκτελεί με τη σειρά αυτό μπορεί να επιστρέψει σωστό αποτέλεσμα. Αν όμως εκτελεστούν με διαφορετική σειρά (π.χ. α, γ, β) τότε μπορεί να προκύψει κάποιο απρόσμενο σφάλμα. Πρακτικά είναι πολύ δύσκολο να ελέγξουμε όλα τα δυνατά μονοπάτια ροής ενός προγράμματος.

Το σημείο εκκίνησης για να χρησιμοποιήσουμε αυτή τη μέθοδο είναι το διάγραμμα ροής του προγράμματος μας. Ένα τέτοιο διάγραμμα θα πρέπει να αποτελείτε από κόμβους που αναπαριστούν αποφάσεις και ακμές που αναπαριστούν την ροή. Το διάγραμμα κατασκευάζεται αντικαθιστώντας την κάθε συνθήκες ελέγχου του προγράμματος με ισοδύναμα διαγράμματα. Κάθε ακμή του διαγράμματος αναπαριστά μια διαφορετική επιλογή σε συνθήκες τύπου if-else ή case. Ένα βέλος που επιστρέφει σε προηγούμενο κόμβο αναπαριστά μια επαναληπτική διαδικασία.



Σχήμα 6: Παράδειγμα ροής για μέθοδο αναζήτησης σε δυαδικό δέντρο(binary tree).

5.1.4 Έλεγχος καταπόνησης (Stress Testing)

Ένας άλλος τρόπος αποδοτικών ελέγχων είναι να χρησιμοποιήσουμε ένα μεγάλο όγκο δεδομένων για είσοδο στο πρόγραμμα με μια αυτοματοποιημένη μέθοδο. Αυτού του είδους οι έλεγχοι εξωθούν πιο εύκολα τα προγράμματα σε οριακές καταστάσεις από ένα τεστ που θα έκανε ο απλός χρήστης. Ο μεγάλος όγκος δεδομένων μπορεί να προκαλέσει προβλήματα υπερχείλισης στο πρόγραμμα που αφορούν την διαχείριση προσωρινής μνήμης, πινάκων και μετρητών. Οι άνθρωποι συνήθως τείνουν να αποφεύγουν τεστ με απίθανες καταστάσεις όπως εισόδους με λάθος δεδομένα, εκτός ορίων ή πολύ μεγάλα σε όγκο. Αντίθετα μια αυτοματοποιημένη μέθοδο μπορεί εύκολα να παράξει είσοδο για το πρόγραμμα χωρίς τους παραπάνω περιορισμούς. Επίσης, οι τυχαίοι αριθμοί για είσοδο είναι ένας άλλος τρόπος να δοκιμάσεις ένα πρόγραμμα με απώτερο σκοπό να το οδηγήσεις σε μια εσφαλμένη κατάσταση.

Μερικά tests αυτού του είδους χρησιμοποιούν ειδικά κακόβουλες εισόδους για να ελέγξουν την ασφάλεια της εφαρμογής. Τέτοιου είδους επιθέσεις χρησιμοποιούν εισόδους μεγάλες σε όγκο ή περιέχουν μη επιτρεπτούς χαρακτήρες. Μερικές συναρτήσεις της standard βιβλιοθήκης είναι επιρρεπής σε τέτοιες επιθέσεις για παράδειγμα η **gets**. Η **gets** δεν παρέχει κάποιο τρόπο για να περιορίσεις την είσοδο από μια γραμμή οπότε θα πρέπει να χρησιμοποιείτε την **fgets**. Η **scanf**,

scanf ("%s", buf)**;**

επίσης δεν έχει κάποιο τέτοιον έλεγχο στο μέγεθος της γραμμής εισόδου οπότε θα πρέπει να χρησιμοποιείτε με αυστηρά όρια κατά την κλήση της. Παράδειγμα:

scanf ("%20s", buf).

5.2 Εσωτερικός έλεγχος (Internal testing)

Η διαφορά με τις παραπάνω μεθόδους που αναφέραμε είναι ότι εδώ αναπτύσσουμε κώδικα για αυτό-έλεγχο του προγράμματος καθώς εκτελείται. Το πρόγραμμα κατά την κατασκευή του θα περιέχει και τα εργαλεία για τον έλεγχο του.

5.2.1 Έλεγχος συνθηκών πριν και μετά την εκτέλεση.

Ένας τρόπος για να προλάβουμε τα προβλήματα είναι να ελέγξουμε ότι οι προαπαιτούμενες ή αναμενόμενες συνθήκες είναι σωστές πριν και μετά την εκτέλεση ενός τμήματος κώδικά. Ένα σύνηθες παράδειγμα είναι να ελέγξουμε τα όρια τιμών στις μεταβλητές εισόδου της συνάρτησης μας.

Χρησιμοποιούμε για παράδειγμα μια συνάρτηση που υλοποιήσαμε στο προηγούμενο κεφάλαιο για να κληρώσουμε τους 6 αριθμούς:

```
/* This functions generates 6 random numbers for a lottery draw
The numbers should be unique and range between 1 and 49 */
void draw_numbers(int draw, TStoixeioyOyras * temp)  {

    int num[6], x, j, i, found = 0, ok = 0;

    temp->lottery_id = draw;

    /*repeat for 6 numbers */
    for(i=0;i<6;i++) {
        unique=0;
        /* repeat untill you get a unique number */
```

```
            while(!unique) {
                /* random number */
                x = rand() % 50 + 1;
                found = 0;
                j = 0;
                while(!found && j < i) {
                    if(x == num[j])  {
                        found = 1;
                    }
                    j++;
                }
                if(!found)  {
                    unique = 1;
                }
                num[i] = x;
                temp->numbers[i] = x;
            }
        }
        return;
}
```

Παρατηρούμε ότι δεν έχει γίνει κανένας έλεγχος για τις τιμές των μεταβλητών που περνάμε σαν παραμέτρους. Για να το διορθώσουμε αυτό θα χρειαστεί να προσθέσουμε τον παρακάτω κώδικά:

```
if(draw < 0 || temp==NULL) {
    printf("\nInvalid data");
    return ;
}
```

Για καλύτερο έλεγχο του αποτελέσματος της συνάρτησης θα χρειαστεί να τροποποιήσουμε τη δήλωση της ώστε να επιστρέφει πάντοτε μια τιμή (κενή (null) αν έγινε κάποιο λάθος ή αν όλα δούλεψαν κανονικά την λίστα των αριθμών που κληρώθηκαν). Έτσι η συνάρτηση μας θα γραφεί ως εξής:

```
/* This functions generates 6 random numbers for a given lottery draw
The numbers should be unique and range between 1 and 49 */
int  * draw_numbers(int draw, TStoixeioyOyras * temp) {

    int num[6], x, j, i, found = 0, ok = 0;

    /* check input variables */
    if(draw < 0 || temp == NULL) {
        printf("\nInvalid data");
        return null;
    }

    temp->lottery_id = draw;

    /*repeat for 6 numbers */
    for(i = 0; i < 6; i++) {
        unique = 0;
        /* repeat until you get a unique number */
        while(!unique) {
            /* random number */
```

```
            x = rand() % 50 + 1;
            found = 0;
            j = 0;
            while(!found && j < i){
                if(x == num[j]){
                    found = 1;
                }
                j++;
            }
            if(!found){
                unique = 1;
            }
            num[i] = x;
            temp->numbers[i] = x;
        }
    }
    return  temp->numbers;
}
```

Χρήση assert

Στη C υπάρχει η δυνατότητα για την χρήση συμβάσεων-συμφωνίων (assertions), προτρέποντας έτσι την προσθήκη ελέγχων για αρχικές και τελικές συνθήκες σε ένα τμήμα του κώδικα. Μια αποτυχημένη κλήση της **assert**() πάντοτε τερματίζει το πρόγραμμα, οπότε προσπαθούμε να την χρησιμοποιούμε μόνο σε ειδικές περιπτώσεις όπου η αποτυχία θα είναι απρόσμενη και το πρόγραμμα δεν θα μπορεί να ανακάμψει μετά από αυτή. Το παραπάνω παράδειγμα θα μπορούσε εύκολα να γραφτεί ως:

```
/* This functions generates 6 random numbers for a given lottery draw
The numbers should be unique and range between 1 and 49 */
int  * draw_numbers(int draw, TStoixeioyOyras * temp) {
    int num[6], x, j, i, found = 0, ok = 0;

    /* assert input variables */
    assert(draw > 0 && temp!=NULL);
}
```

Όπου ορίζουμε σαν περιορισμό ότι ο αριθμός της κλήρωσης (draw) πρέπει να είναι μη αρνητικός και ότι ο δείκτης για το στοιχειό (temp) να είναι έγκυρος. Σε περίπτωση που ο έλεγχος της **assert** αποτύχει περιμένουμε το πρόγραμμα να τερματίσει με ένα μήνυμα λάθους : **Assertion violation**

Η χρήση της **assert** βοηθά επίσης στο να επαληθεύουμε τις ιδιότητες των διεπαφών επειδή κάνουν πιο ξεκάθαρες τις ασυμφωνίες μεταξύ των προγραμμάτων που χρησιμοποιούν μια διεπαφή και τις ίδιας της συνάρτησης. Στο παραπάνω παράδειγμα αν η assert αποτύχει γιατί η τιμή του **draw** είναι αρνητική, θα σημάνει ότι καλούμε την συγκεκριμένη συνάρτηση με λάθος τρόπο. Αν λοιπόν αργότερα αλλάξουμε τη συμπεριφορά μιας διεπαφής και ξεχάσουμε να αλλάξουμε και τον τρόπο κλήσης της σε κάποιο σημείο του κώδικα, τότε με την **assert** θα μπορέσουμε εύκολα να εντοπίσουμε το λάθος πριν προκαλέσει σοβαρότερα προβλήματα αργότερα.

Η **assert** θα μπορούσε να μας φανεί χρήσιμη και στον έλεγχο του αποτελέσματος της συνάρτησης. Με μια τέτοια εντολή για παράδειγμά:

```
assert(temp->number[1] > 0 && temp->number[1] < 50)
```

Θα μπορούσαμε να εξασφαλίσουμε ότι οι συνθήκες μετά την εκτέλεση της συνάρτησης εξακολουθούν να παραμένουν σωστές. Έστω ότι χρειάστηκε να αλλάξουμε ένα τμήμα κώδικα από τη συνάρτηση και κατά την μετέπειτα εκτέλεση προκύψει ένα **assesrion violation**, τότε θα είμαστε σίγουροι ότι το λάθος μας προέκυψε από την ίδια την συνάρτηση και όχι από το τον υπόλοιπο κώδικά που την καλεί.

Ένα θέμα που ενδεχομένως θα μπορούσε να προκύψει με την χρήση assertions είναι ότι όταν τελειώσουμε οριστικά με την κατασκευή και τον έλεγχο του προγράμματος θα θέλαμε να αφαιρέσουμε τα assertions από τον κώδικα μας πριν τον παραδώσουμε στον πελάτη. Ένας τρόπος φυσικά θα ήταν να σβήσουμε όλες τις εντολές assert από τον κώδικα. Αυτό όμως δεν είναι καλή πρακτική γιατί έτσι χάνουμε χρήσιμες πληροφορίες για τις αναμενόμενες λειτουργείς και τα όρια της κάθε συνάρτησης. Ένας καλύτερος τρόπος είναι με τη χρήση της μακροεντολής **NDEBUG**. Πρακτικά όταν δηλώνουμε την μακροεντολή αυτή αμέσως μετά την τελευταία δήλωση για την assert.h τότε η μακροεντολή assert δεν δημιουργεί κανένα επιπλέον κώδικά και δεν εκτελεί καμία ενέργεια.

5.2.2 Έλεγχος τιμών επιστρεφόμενων λαθών

Κάτι που παραλείπεται συχνά από ένα πρόγραμμα είναι ο έλεγχος των λαθών που επιστρέφουν οι συναρτήσεις βιβλιοθήκης και οι κλήσεις συστήματος. Οι τιμές από συναρτήσεις εισόδου, όπως η **fread** και η **fscanf**, πρέπει πάντοτε να ελέγχονται για λάθη, όπως και οι κλήσεις της **fopen**. Αν μια τέτοια κλήση αποτύχει πιθανώς δεν πρέπει να συνεχίσουμε στο υπόλοιπο πρόγραμμα. Όταν λοιπόν, ελέγχουμε τις τιμές που επιστρέφουν κλήσεις όπως η **fprintf** και η **fwrite** μπορούμε να εντοπίσουμε σφάλματα γραφής στα αρχεία όπως για παράδειγμα ότι δεν έχει άλλο ελεύθερο χώρο. Ένας τυπικός έλεγχος για την **fopen**:

```
if( (fp=fopen(filename,"r")) == NULL ) {
    printf("\nFile %s not found\n",filename);
    exit(1);
}
```

Ο επιπλέον φόρτος για να προσθέσεις τέτοιους ελέγχους είναι μηδαμινός ενώ τα οφέλη είναι μεγάλα. Όταν σκεφτόμαστε τους ελέγχους κατά της διάρκεια του γραψίματος οδηγούμαστε να γράψουμε καλύτερο κώδικα, διότι αυτή είναι η καλύτερη στιγμή και γνωρίζουμε ακριβώς τι κάνει το συγκεκριμένο κομμάτι. Αν αντιθέτως περιμένουμε μέχρι κάτι να χαλάσει, το πιο πιθανό είναι να μην θυμόμαστε τότε τι κάνει το συγκεκριμένο κομμάτι κώδικα. Έτσι θα πρέπει να εργαστούμε υπό πίεση και να αναλώσουμε περισσότερο χρόνο μέχρι να λύσουμε το πρόβλημα χωρίς να είμαστε απολύτως σίγουροι για την αξιοπιστία και την ανθεκτικότητα της λύσης μας.

5.2.3 Προσωρινές αλλαγές στον κώδικα

Για τη μέθοδο αυτή αλλάζουμε τον κώδικα του προγράμματος μας προσωρινά για να εισάγουμε τεχνητά όρια ή να αλλάξουμε συνθήκες με σκοπό την υπερφόρτωσης του προγράμματος μας. Για να παράδειγμα σε ένα πρόγραμμα ταξινόμησης αριθμών με τη χρήση πινάκων θα μπορούσαμε να μειώσουμε αρκετά τα όρια του κάθε πίνακα. Με αυτό τον τρόπο θα μπορούσαμε να ελέγξουμε εύκολα την απόδοση της εφαρμογής για σχετικά μεγάλα σύνολα αριθμών όπως και τα πιθανά σενάρια υπερχείλισης των δομών μας.

Σαν ένα άλλο παράδειγμα θα μπορούσαμε με αυτή τη μέθοδο να ελέγξουμε μια εφαρμογή που χρησιμοποιεί πίνακές κατακερματισμού. Θα μπορούσαμε για

παράδειγμα να αλλάξουμε την συνάρτηση κατακερματισμού (hash function) ώστε να επιστέφει πάντοτε μια σταθερή τιμή. Με αυτό τον τρόπο όλα τα δεδομένα θα καταλήξουν στον ίδιο κάδο οπότε και θα προκαλέσουμε πολύ εύκολα υπερχείλιση. Επίσης θα μπορέσουμε να ελέγξουμε εύκολα την απόδοση τη εφαρμογής για τις διαδικασίες της εισαγωγής και της αναζήτησης σε πολύ μεγάλους κάδους.

5.2.4 Αφήνουμε ανέπαφο τον κώδικα ελέγχου

Μια άλλη μέθοδος ελέγχου είναι να συμπεριλάβουμε τον κωδικά ελέγχου ανέπαφο μαζί με το κυρίως πρόγραμμα μας. Αυτό θα μπορούσαμε να το πετύχουμε εύκολα περιβάλλοντας τον κώδικα μας με την μακροεντολή **NDEBUG**. Για παράδειγμα :

```
#ifndef NDEBUG
int isValid(MyType object) {
…
Test invariants here.
Return 1 (TRUE) if object passes
all tests, and 0 (FALSE) otherwise.
…
}
#endif
void myFunction(MyType object) {
assert(isValid(object));
…
Manipulate object here.
…
assert(isValid(object));
}
```

Αυτό θα είχε σαν αποτέλεσμα να κάνουμε το πρόγραμμα μας πιο εύκολο στη συντήρηση καθώς μετά από κάθε αλλαγή θα είναι εξίσου εύκολο να ελέγξουμε το πρόγραμμα με τις προ υπάρχουσες δομές ελέγχου. Από την άλλην όμως όταν χρησιμοποιούμε εκτενώς συναρτήσεις ελέγχου στο πρόγραμμα μας μπορεί να κάνουμε πιο δύσκολη τη συντήρηση του καθώς με κάθε αλλαγή θα πρέπει να ενημερώνουμε και να συντηρούμε επαρκώς και τον επιπλέον κώδικα που κάνει τον έλεγχο.

5.3 General testing strategies

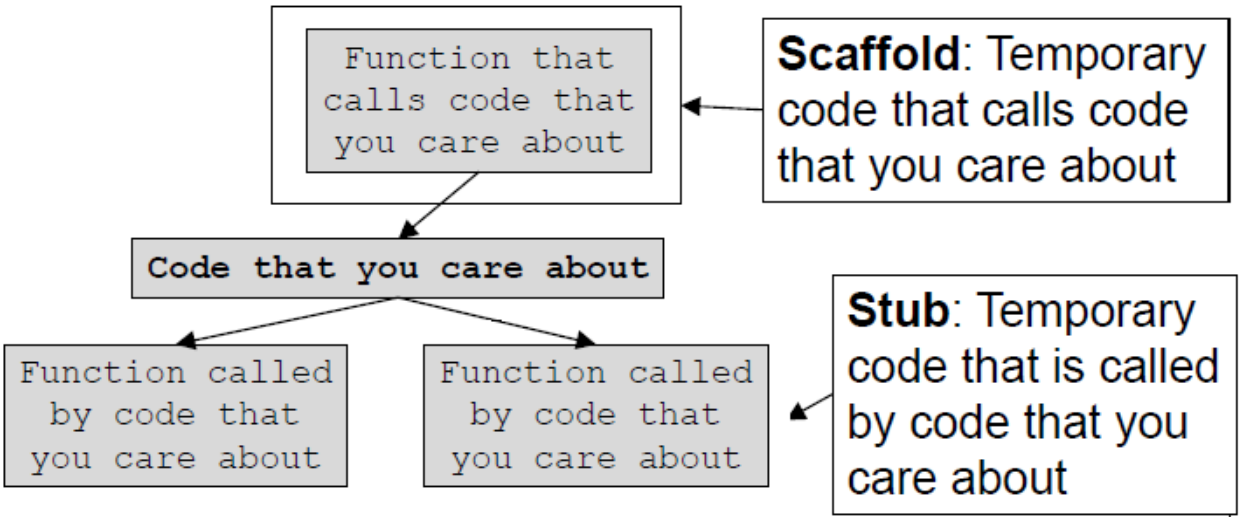
Ακολουθούν κάποιες γενικές μέθοδοι ελέγχου που μπορούν να σας φανούν χρήσιμες σε όλες τις περιπτώσεις.

5.3.1 Test Scaffolds

Σε πολλές περιπτώσεις όταν ελέγχεις ένα μεγάλο πρόγραμμα χρειάζεται να δημιουργήσεις κάποιο είδος πλαισίου (framework) ή scaffold για να μπορέσεις να ελέγξεις ένα μεμονωμένο τμήμα ξεχωριστά. Αυτό το πλαίσιο θα παρέχει μια διεπαφή στο υπόλοιπο σύστημα κάτω από το οποίο θα ελέγξεις το συγκεκριμένο τμήμα του κώδικα.

Είναι εύκολο να κατασκευάζεις scaffolds για τον έλεγχο αριθμητικών συναρτήσεων, συναρτήσεων επεξεργασίας συμβολοσειρών, συναρτήσεων ταξινόμησης και άλλων συναρτήσεων. Η διαδικασία συνήθως αποτελείται από το να ορίζεις τις παραμέτρους εισόδου, να καλείς τις συναρτήσεις που θα ελεγχθούν και στο τέλος να ελέγχεις τα

αποτελέσματα. Είναι πιο δύσκολο όμως να δημιουργείς scaffolds για μεγαλύτερα προγράμματα που είναι ακόμα ημιτελή.



Σχήμα 7: Γενική αναπαράσταση scaffold και stub σε ένα πρόγραμμα.

Αυτό μπορεί να γίνει πιο κατανοητό στο παρακάτω παράδειγμα, όπου προσπαθούμε να κατασκευάσουμε μερικούς ελέγχους για την εντολή memset (μια από τις συναρτήσεις της standard βιβλιοθήκης της C/C++). Η συνάρτηση αυτή είναι συνήθως γραμμένη σε γλώσσα assembly για καλύτερες επιδόσεις. Επειδή είναι προσεκτικά βελτιστοποιημένη όμως είναι πιο πιθανό να παρουσιάσει λάθη και για αυτό πρέπει να ελεγχθεί εκτενέστερα.

Το πρώτο βήμα είναι να κατασκευάσουμε την πιο απλή εκδοχή που να δουλεύει σωστά. Αυτή μπορεί να χρησιμοποιηθεί μετά σαν κριτήριο επίδοσης και ορθότητας των αποτελεσμάτων.

Έστω η συνάρτηση memset (s, c, n) η οποία θέτει σε n bytes μνήμης την τιμή c αρχίζοντας από την διεύθυνση s και επιστρέφει το s. Στην πιο απλή μορφή θα μπορούσε να γραφτεί ως:

```
/* memset: set first n bytes of s to c */
void *memset(void *s, int c, size_t n) {
    size_t i;
    char *p;

    p = (char *)s;
    for (i = 0; i < n; i++) {
        p[i] = c;
    }
    return s;
}
```

Όταν όμως έχουμε θέμα με την απόδοση της συγκεκριμένης συνάρτησης τότε μπορούμε να χρησιμοποιήσουμε τεχνική για να γράψουμε πλήρες λέξεις (32 ή 64 bits) σε κάθε βήμα τις επανάληψης. Αυτές οι τεχνικές όμως μπορούν να οδηγήσουν σε σφάλματα για αυτό είναι υποχρεωτικοί οι εκτενής έλεγχοι. Οι έλεγχοι θα βασιστούν σε οριακές συνθήκες και πιθανών καταστάσεων αποτυχίας.

Για την συνάρτηση memtest τα όρια περιλαμβάνουν τιμές όπως n = 0, 1, 2 αλλά και τιμές που είναι δυνάμεις του 2 συμπεριλαμβανομένων και αυτών των αριθμών που είναι κοντά σε αυτές τις τιμές. Επίσης πρέπει να ελεγχθούν οι τιμές που είναι κοντά στα

φυσικά όρια που μπορεί να χρησιμοποιήσει ένας υπολογιστής όπως 2^{16} σε υπολογιστές με λέξεις μεγέθους 16 bit. Ειδικά για οι δυνάμεις του 2, χρήζουν προσοχής μιας και ένας τρόπος να επιταχύνεις την memset είναι να αναθέτεις πολλά byte μαζί, συνήθως πολλαπλάσια του 2. Με παρόμοιο τρόπο, θέλουμε να ελέγξουμε τις αρχικές καταστάσεις του πίνακά μας μιας και μπορούν να προκύψουν λάθη με την αρχική διεύθυνση s ή το μήκος του. Αυτό μπορούμε να το πετύχουμε τοποθετώντας τον πίνακα μας μέσα σε ένα μεγαλύτερο πίνακα. Με αυτόν τον τρόπο δημιουργούμε μια ζώνη προστασίας. Τέλος θέλουμε να ελέγχουμε και μια πληθώρα διαφορετικών τιμών c, συμπεριλαμβανομένων του 0 και του 0x7F (του μεγαλύτερου προσημασμένου αριθμού για 8bits) όπως και αριθμών μεγαλύτερων του ενός byte για να είμαστε σίγουροι ότι μόνο το πρώτο byte του αριθμού θα χρησιμοποιηθεί. Εδώ θα βοηθήσει να αρχικοποιήσουμε την μνήμη μας με μια διαφορετική και χαρακτηριστική τιμή ώστε μετά να μπορούμε να ελέγξουμε αν η memset έγραψε εντός του ορίου n που έπρεπε.

Μπορούμε να χρησιμοποιήσουμε την παραπάνω απλή υλοποίηση μας σαν ένα πρότυπο σύγκρισης με την γρήγορη μέθοδο που θέλουμε να ελέγξουμε. Στο παρακάτω test θα δεσμεύσουμε χώρο για 2 πίνακες και μετά θα συγκρίνουμε τη συμπεριφορά τους σε συνδυασμούς από n και c τιμών με την εξής λογική:

```
big = maximum left margin + maximum n + maximum right margin
s0 = malloc(big)
sl = malloc(big)
for each combination of test parameters n, c, and offset:
    set all of s0 and s1 to known pattern
    run slow memset(s0 + offset, c, n)
    run fast memset(s1 + offset, c, n)
    check return values
    compare all of s0 and sl byte by byte
```

Ένα λάθος που θα έκανε την memset να γράψει εκτός ορίων του πίνακα είναι πιο πιθανό ότι θα επηρεάσει τα bytes στην αρχή ή στο τέλος του πίνακα. Έτσι δεσμεύοντας παραπάνω χώρο για μια ζώνη προστασίας (buffer zone) μπορούμε να εντοπίσουμε τα προβληματικά bytes και να εξασφαλίσουμε ότι το λάθος δεν θα επηρεάσει κάποια άλλη σημαντική μνήμη του προγράμματος. Σε τέτοιες περιπτώσεις πρέπει να ελέγξουμε όλα τα bytes του s0 και s1 και όχι μόνο τα n που θα έπρεπε να γραφούν. Μερικές επιθυμητές τιμές ελέγχου είναι οι εξής :

```
offset = 10, 11, ..., 20
c = 0, 1, Ox7F, 0x80, OxFF, Ox11223344
n= 0,1,2,3,4,5,7,8,9,15,16,17,31, 32, 33, ..., 65535, 65536, 65537
```

Οι τιμές του n θα πρέπει να συμπεριλαμβάνουν τουλάχιστον τους αριθμούς 2^x, $2^x + 1$, $2^x - 1$ όπου x = 0..16. Αυτές οι τιμές δεν θα πρέπει να είναι ορισμένες αυστηρά μέσα στο κυρίως σώμα του test scaffold αλλά θα πρέπει να δημιουργούνται δυναμικά ανά περίπτωση.

Ένα τέτοιο είδος ελέγχου κοστίζει πολύ λίγο χρόνο αν εκτελεστεί και διατρέχει ένα μεγάλο πλήθος τιμών ώστε να εξασφαλίζει διεξοδικό έλεγχο της συνάρτησης. Επίσης μπορεί να μεταφερθεί εύκολα σε ένα νέο περιβάλλον όπου πιθανώς θα χρησιμοποιεί διαφορετικά μεγέθη λέξεων.

5.3.2 Έλεγχος στα απλά τμήματα πρώτα.

Η τμηματική προσέγγιση εφαρμόζεται στο πώς θα ελέγχεις τα επιμέρους στοιχεία του προγράμματος. Ο έλεγχος θα πρέπει να εστιάζει πρώτα στα απλά και πιο συχνά

χρησιμοποιημένα χαρακτηριστικά. Μόνο όταν σιγουρευτούμε ότι αυτά δουλεύουν κανονικά θα πρέπει να προχωρήσουμε και στα πιο σύνθετα. Τα εύκολα τεστ βρίσκουν και τα πιο συνηθισμένα σφάλματα. Κάθε τεστ βοηθά, έστω και στο ελάχιστο να ξετρυπώσουμε τα ενδεχόμενα προβλήματα που θα συναντήσουμε.

Σε αυτό το τμήμα θα αναφερθούμε σε τρόπους για να επιλέξουμε τους πιο αποτελεσματικούς ελέγχους και με πια σειρά θα πρέπει να τους εκτελέσουμε. Το πρώτο βήμα, τουλάχιστον για μικρές συναρτήσεις είναι μια προέκταση του έλεγχου οριακών τιμών που αναφερθήκαμε πριν. Αν για παράδειγμα έχουμε μια συνάρτηση που υλοποιεί μια δυαδική αναζήτηση σε ένα πίνακα ακεραίων, θα ξεκινούσαμε με τους παρακάτω ελέγχους:

- Αναζήτηση σε ένα άδειο πίνακα.
- Αναζήτηση σε πίνακα με ένα μόνο στοιχείο όπου η τιμή ελέγχου θα είναι:
  1. μικρότερη από αυτή του στοιχείου.
  2. ίση με αυτή του στοιχείου.
  3. μεγαλύτερη από αυτή του στοιχείου.
- Αναζήτηση σε ένα πίνακα με 2 στοιχεία και τιμές που θα περιλαμβάνουν και τους
5 τους πιθανούς συνδυασμούς.
- Έλεγχος συμπεριφοράς διπλότυπων τιμών στον πίνακα με τιμής ελέγχου:
  1. μικρότερη από την τιμή του στοιχείου.
  2. ίσες.
  3. μεγαλύτερες.
- Αναζήτηση ενός πίνακα με 3 στοιχεία όπως έγινε παραπάνω με τα 2 στοιχειά.
- Αναζήτηση έως πίνακα με 4 στοιχειά όπως και παραπάνω.

Αν μέχρι αυτό το σημείο δεν έχουν προκύψει προβλήματα είναι πολύ πιθανό ότι η συνάρτηση είναι λειτουργική αλλά μπορούμε να κάνουμε και επιπλέον ελέγχους.

Αν και αυτοί οι έλεγχοι είναι σχετικά μικροί και μπορούν να γίνουν χειροκίνητα είναι πιο εύκολο να κατασκευάσουμε ένα μηχανισμό (test scaffold) για να αυτοματοποιήσουμε τη διαδικασία. Το παρακάτω απλό πρόγραμμα διαβάζει 2 αριθμούς από την είσοδο, ένα κλειδί για αναζήτηση και το μέγεθος του πίνακα, και κατασκευάζει ένα πίνακα που περιέχει τιμές 1, 3, 5, … στον οποίο ψάχνει το δεδομένο κλειδί.

```
int main(void)
{
    int i , key, nelem, arr[1000];

    while (scanf("%d %d" , &key, &nelem)!= EOF) {
        for (i = 0; i < nelem; i++)  {
            arr[i] = 2*i + 1;
        }
        printf("%d\n" , binsearch(key, arr, nelem));
    }
    return 0;
}
```

Αν και απλοϊκό πρόγραμμα, δείχνει πόσο χρήσιμος είναι ένας τέτοιος μηχανισμός που μπορεί εύκολα να επεκταθεί για να καλύψει περισσότερους ελέγχους και να απαιτεί μικρότερή ανθρώπινη παρέμβαση.

5.3.3 Bottom-up Testing

Παρά το γεγονός ότι μπορεί να έχετε χρησιμοποιήσει ιεραρχικό (top-down) σχεδιασμό με σταδιακό καθαρισμό (stepwise refinement) σαν στρατηγική για να φτάσετε στον

λεπτομερή σχεδιασμό και στην υλοποίηση του προγράμματος σε C κώδικα, μπορείτε να χρησιμοποιήσετε την αντίθετη πορεία για να το ελέγξετε. Αυτό σημαίνει ότι έχει νόημα να προχωρήσετε bottom-up (από κάτω προς τα πάνω) για ένα πιο αποτελεσματικό έλεγχο.

Υποθέστε ότι πρώτα προσπαθείτε να αναγνωρίσετε τα κομμάτια του προγράμματός σας που αποτελούν το κατώτερο σημείο του. Έστω ένα σύνολο συναρτήσεων S1 οι οποίες δεν καλούν καμία άλλη συνάρτηση. Αυτές είναι οι κατώτερες συναρτήσεις μέσα στο σύστημά σας (μιας και δεν καλούν καμία άλλη συνάρτηση σε κατώτερο επίπεδο). Καταρχάς, μπορείτε να ξεκινήσετε να ελέγχετε εξονυχιστικά αυτές τις συναρτήσεις, για να βεβαιωθείτε ότι λειτουργούν σωστά. Τώρα είσαστε σίγουροι ότι οι συναρτήσεις του S1 λειτουργούν σωστά και είναι κατάλληλες για χρήση στις επόμενες ενέργειές σας.

Στη συνέχεια αναγνωρίζετε ένα επόμενο στρώμα/επίπεδο συναρτήσεων S2, οι οποίες καλούν μόνο τις συναρτήσεις του S1(που δεν καλούν άλλες συναρτήσεις). Στη συνέχεια ελέγχετε τις συναρτήσεις του S2. Ακολουθώντας αυτή την πορεία βρίσκετε το επόμενο επίπεδο των συναρτήσεων S3, που καλούν τις ήδη ελεγμένες συναρτήσεις του S1 ή του S2. Συνεχίζοντας με την ίδια τακτική, σε κάθε στάδιο βρίσκετε ένα νέο επίπεδο συναρτήσεων Sn, οι οποίες καλούν τις ήδη ελεγμένες συναρτήσεις και αυξάνεται το πλήθος των ελεγμένων τμημάτων του συστήματός σας, ελέγχοντας τις συναρτήσεις του Sn. Τελικά θα φτάσετε στο ανώτερο επίπεδο του προγράμματός σας, το κυρίως πρόγραμμα (main) του συστήματός σας και θα το ελέγξετε.

Στον από κάτω προς τα πάνω έλεγχο, μερικές φορές η σειρά με την οποία ελέγχονται οι συναρτήσεις επηρεάζεται:

1. από τον τρόπο με τον οποίο οι συναρτήσεις χειρίζονται τα δεδομένα και
2. με τη σειρά με την οποία γίνονται οι κλήσεις των συναρτήσεων.

Για παράδειγμα, θα πρέπει να ελέγξετε την συνάρτηση αρχικοποίησης ενός πίνακα (ή λίστας), πριν από τον έλεγχο της συνάρτησης λειτουργίας του πίνακα (ή της λίστας). Προϋπόθεση εδώ είναι ότι ο πίνακας (ή λίστα) έχει ήδη αρχικοποιηθεί.

Αρχίζοντας τον bottom-up έλεγχο ελέγχουμε όλες τις συναρτήσεις που καλεί μια συνάρτηση ή αυτές που δίνουν δεδομένα που χρησιμοποιούνται από την συνάρτηση, οποτεδήποτε είναι δυνατό, προτού να ελέγξετε την ίδια την συνάρτηση.

Χρησιμοποιώντας το παράδειγμα από το κεφάλαιο 6 θα μπορούσαμε να χαρακτηρίσουμε συναρτήσεις επιπέδου S1 τις παρακάτω βοηθητικές συναρτήσεις του τύπου στοιχείου ουράς.

```
void StoreElement(QueueElement *target,int *nums,float amount) {
    int i = 0;

    target->customer_id = nums[0];
    target->lottery_id = nums[1];

    for(i=0;i<6;i++) {
        target->numbers[i] = nums[i+2];
    }
    target->amount = amount;

    return;
}

void ElementGetNumbers(QueueElement source,int **nums) {
    *nums = source.numbers;
```

```
    return;
}
```

Ένας πολύ εύκολος τρόπος να ελέγξουμε την ορθότητα τους είναι με το παρακάτω υπό πρόγραμμα:

```
#include <stdio.h>
#include <stdlib.h>
#include "QueueElement.h"

int main() {
    QueueElement  * new_queue;

    int *nums;
    int *result_nums;
    float amount;
    int i;

    new_queue = (QueueElement  *)malloc(sizeof(QueueElement));
    nums = (int *)malloc(8*sizeof(int));
    result_nums = (int *)malloc(6*sizeof(int));

    printf("Generate rundom numbers\n");
    for(i = 0; i < 8; i++) {
        nums[i] = rand() % 50;
    }
    amount = (float)((rand()  %100) / 10);
    printf("Call : StoreElement\n");
    StoreElement(new_queue,nums,amount);
    printf("Call : ElementGetNumbers\n");

    printf("Test results\n");
    for(i = 2; i < 8; i++) {
        if(nums[i]! = result_nums[i-2]) {
            printf("Error: numbers don't match (i=%d): %d -
                    %d\n",i,nums[i],result_nums[i]);
            return -1;
        }
    }

    printf("Succesfull test!\n");
    return 0;
}
```

Ο έλεγχος μιας συγκεκριμένης συνάρτησης ονομάζεται unit testing (Ο όρος unit testing προέρχεται από το βιομηχανικό χώρο των μηχανικών λογισμικού). Στο unit testing λοιπόν ασχολούμαστε με τα συγκεκριμένα χαρακτηριστικά που είναι απαραίτητα για την λειτουργία της εκάστοτε συνάρτησης. Μόλις ολοκληρωθούν όλοι οι έλεγχοι επιτυχώς τότε μπορούμε να ασχοληθούμε και με τα μεγαλύτερα τμήματα του προγράμματος.

Με αυτό τον τρόπο μπορούμε να εξασφαλίσουμε ότι και οι 2 συναρτήσεις λειτούργησαν σωστά με μια οποιαδήποτε τυχαία είσοδο. Φυσικά για να έχουμε ένα πιο πλήρες σύνολο από δοκιμές θα πρέπει να εφαρμόσουμε και όσα αναφέρθηκαν στις προηγούμενες ενότητες. Δηλαδή, θα μπορούσαμε να ελέγξουμε τις συναρτήσεις μας με

οριακές τιμές ή θα μπορούσαμε να προσθέσουμε επιπλέον assertions για τον έλεγχο των αρχικών και τελικών αριθμών σε κάθε κλήση. Για παράδειγμα, στη συνάρτηση StoreElement θα μπορούσαμε να προσθέσουμε τον έλεγχο:

```
assert(target != NULL && nums != NULL);
```

Αυτό θα μας εξασφάλιζε ότι σε περίπτωση μη έγκυρης κλήσης της συνάρτησης, θα ήταν εύκολο να εντοπίσουμε το πρόβλημα στον κώδικα που θα την καλούσε. Το παραπάνω παράδειγμα μπορούμε να το επεκτείνουμε για και ελέγξουμε όλες τις συναρτήσεις του πρώτου επιπέδου του QueueElement.

Στη συνέχεια, έχοντας εξασφαλίσει ότι όλες οι συναρτήσεις του πρώτου επιπέδου είναι σωστές μπορούμε να προχωρήσουμε στο δεύτερο επίπεδο S2. Στις συναρτήσεις δηλαδή που χρησιμοποιούν όλες τις κλήσεις του QueueElement όπως:

```
void InsertQueue(const QueueHandle QueuePtr, QueueElement * const ElementPtr,
                 int *overflow){
    /*  Pre:    Existing queue
       After:   New element is pushed in the queue    */
    if (IsFullQueue(QueuePtr)){
        *overflow = 1;
    }
    else{
        *overflow = 0;
        QueuePtr->metritis++;
        ElementSetValue (&(QueuePtr->pinakas[QueuePtr->piso]), *ElementPtr);
        QueuePtr->piso = ( QueuePtr->piso + 1 ) %  PLITHOS;
    }
}
```

Από την περιγραφή της παραπάνω συνάρτησης μπορούμε να συμπεράνουμε ότι απαραίτητη προϋπόθεση είναι η δημιουργία μιας στοίβας. Αυτό συνεπάγεται ότι τη συνάρτηση δημιουργίας της στοίβας θα πρέπει να την εντάξουμε στις συναρτήσεις του επιπέδου S1. Οπότε και οι αρχικοί μας έλεγχοι θα πρέπει να συμπεριλάβουν και αυτή την συνάρτηση. Η διαδικασία θα ολοκληρωθεί μόλις τελειώσουμε τους ελέγχους στις συναρτήσεις σε όλα τα επίπεδα και καταλήξουμε στο κυρίως πρόγραμμα (συνάρτηση main).

5.3.4 Σύγκριση ανεξάρτητων υλοποιήσεων

Οι ανεξάρτητες υλοποιήσεις μιας βιβλιοθήκης ή ενός προγράμματος θα πρέπει να παράγουν τα ίδια αποτελέσματα. Για παράδειγμά δυο μεταγλωττιστές θα πρέπει να παράγουν προγράμματα που συμπεριφέρονται με το ίδιο τρόπο στο ίδιο μηχάνημα, τουλάχιστον στις περισσότερες περιπτώσεις.

Σε μερικές περιπτώσεις ένα αποτέλεσμα μπορεί να υπολογιστεί με παραπάνω από ένα τρόπους. Επίσης σε κάποιες περιπτώσεις μπορείς να γράψεις μια πιο απλή υλοποίηση για το ίδιο πρόβλημα χωρίς να λογαριάζεις την απόδοση. Αν δυο ασύνδετα προγράμματα που υλοποιούν την ίδια λειτουργικότητα, παράγουν το ίδιο αποτέλεσμα τότε υπάρχει μεγάλη πιθανότητα να είναι σωστά. Ενώ αν έχουν διαφορετικά αποτελέσματα τότε τουλάχιστον κάποιο από τα 2 είναι λάθος.

5.3.5 Automation (Αυτοματισμοί)

Είναι προφανές ότι είναι πολύ κουραστικό και μερικές φορές επισφαλής το να κάνουμε όλους τους ελέγχους με το χέρι. Ένα σωστό σύνολο από ελέγχους περιλαμβάνει μεγάλο

πλήθος και πολύπλοκους ελέγχους, με πολλές διαφορετικές εισόδους και πολλές διαφορετικές συγκρίσεις αποτελεσμάτων. Οπότε σίγουρα αξίζει να αφιερώσουμε χρόνο για να γράψουμε ένα απλό πρόγραμμα που θα συμπεριλαμβάνει όλους τους ελέγχους μας. Με αυτό τον τρόπο μπορούμε να εκτελούμε όλους τους ελέγχους μας άμεσα, γρήγορα και χωρίς μεγάλη σπατάλη χρόνου.

5.3.5.1 Regression Testing

Μια κατηγορία ελέγχων που μπορεί να αυτοματοποιηθεί είναι και τα regression tests. Εφόσον το σύστημα παραδοθεί στον πελάτη για λειτουργία, μπαίνει στη φάση συντήρησης (maintenance) του κύκλου ζωής του λογισμικού του. Κατά τη διάρκεια αυτής της περιόδου, ανακαλύπτονται λάθη τα οποία διορθώνονται και αναπόφευκτα το σύστημα αναβαθμίζεται για να μπορεί να εκτελεί νέες συναρτήσεις οι οποίες εξασφαλίζουν ότι το σύστημα θα παραμείνει χρήσιμο προς τους χρήστες του. Μετά από τη διόρθωση ενός λάθους ή την αναβάθμιση του συστήματος ώστε να κάνει κάτι καινούριο, είναι πολύ σημαντικό να ελέγξουμε ότι το λάθος όντως διορθώθηκε και ότι το καινούριο χαρακτηριστικό (feature) λειτουργεί. Επίσης πρέπει να ελέγξουμε πάλι ότι όλα τα προϋπάρχοντα χαρακτηριστικά του προγράμματος λειτουργούν σωστά. Τα regression tests χρησιμοποιούνται για να βεβαιωθούμε ότι όλα όσα λειτουργούσαν σωστά πριν την αλλαγή, συνεχίζουν να λειτουργούν σωστά και μετά από την εγκατάσταση αυτής της αλλαγής μέσα στο σύστημα.

Συνήθως ακολουθούμε τα εξής τέσσερα βήματα όταν γίνεται μία σειρά από tests κατά τη διάρκεια του regression testing:

1 βρίσκουμε ένα λάθος.
2 διορθώνουμε το λάθος.
3 προσθέτουμε κάποιο καινούριο test που ελέγχει αν η αλλαγή που κάναμε διορθώνει το λάθος.
4 τρέχουμε τη σειρά των tests.

Θα ξαφνιαστείτε αν δείτε πόσο πολύ το regression testing ελαχιστοποιεί τα λάθη που υπάρχουν στο σύστημα ή που θα εμφανιστούν με την εισαγωγή ενός καινούριου χαρακτηριστικού, πριν το προϊόν δοθεί στον πελάτη. Αυτή η τακτική είναι πολύ χρήσιμη γιατί βοηθά του μηχανικούς λογισμικού να αποφεύγουν τη δυσαρέσκεια των χρηστών, από τυχόν δυσλειτουργία του προϊόντος.

Είναι προφανές ότι για να είναι αποτελεσματικό το regression testing προϋποθέτουμε ότι η αρχική έκδοση του προγράμματος περνάει όλους τους ελέγχους με επιτυχία. Μόνο αυτό μας εξασφαλίζει ότι ένα επιτυχές αποτέλεσμα στη νέα έκδοση του προγράμματος θα είναι πραγματικά σωστό. Αν κάποια στιγμή από απροσεξία εισάγουμε ένα λανθασμένο έλεγχο τότε θα είναι πολύ δύσκολο να τον εντοπίσουμε στις μετέπειτα φάσεις του προγράμματος. Για αυτό είναι καλή πρακτική να ελέγχουμε τακτικά τα ίδια τα τεστ που χρησιμοποιούμε.

5.3.5.2 Integration Testing (έλεγχοι ενσωμάτωσης)

Στη φιλοσοφία του από κάτω προς τα πάνω ελέγχου, το unit testing γίνεται πρώτα για να ελέγξουμε την συμπεριφορά των συναρτήσεων μέσα στο σύστημα. Στη συνέχεια όμως έχουμε το integration testing. Στο integration testing, ελέγχουμε αν όλες οι συναρτήσεις λειτουργούν σωστά μαζί. Χρησιμοποιώντας Test drivers / test scaffolds μπορούμε να έχουμε μία σειρά σεναρίων για να ελέγξουμε τη σωστή λειτουργία ενός συστήματος. Ένας test driver T, για μία συνάρτηση S, είναι μία συνάρτηση που περιέχει σενάρια ελέγχου για την S, καλεί την S και για κάθε περίπτωση ελέγχει τις

τιμές εισόδου και εξόδου για να δει αν ήταν σωστές. Ένα παράδειγμα test driver είναι και η παρακάτω συνάρτηση:

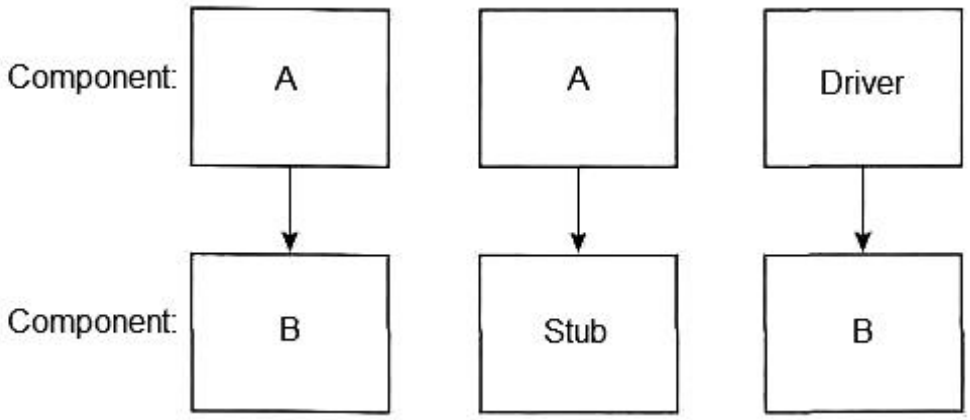
```
int test_TSoyra_getNumbers() {
    TStoixeioyOyras demo;

    int * result;
    int i;

    /* initialize test variables */
    result = (int *)malloc(6*sizeof(int));
    for( i = 0; i < 6 ; i++){
        demo.numbers[i] = rand();
    }
    /* call function */
    TSoyra_getNumbers(demo,&result);
    /* check results */
    for( i = 0; i < 6 ; i++) {
        if(demo.numbers[i] != result[i]) {
            printf("test_TSoyra_getNumbers:Test failed!");
            return -1;
        }
    }
    printf("test_TSoyra_getNumbers:Test successful!");
    return 0;
}
```

Σε ορισμένες περιπτώσεις, αυτή η σειρά σεναρίων μπορεί να αυτοματοποιηθεί, χωρίς να χρειάζεται ανθρώπινη παρέμβαση.

Σε αντίθεση με τα test drivers υπάρχουν περιπτώσεις που χρειάζεται να ελέγξουμε συναρτήσεις όπου κάποια τμήματα τους δεν έχουν ακόμα ολοκληρωθεί. Εδώ μας είναι χρήσιμη η έννοια του stub. Ένα stub είναι μία συνάρτηση που μπαίνει στη θέση μίας άλλης συνάρτησης (που θα γραφτεί αργότερα) και «πλαστογραφεί» τις εξόδους της σε συγκεκριμένα σενάρια ελέγχου. Η μέθοδος αυτή προφανώς έχει μεγάλη χρησιμότητα στον top-down προγραμματισμό.



Σχήμα 8: Διαφορά μεταξύ stub και scaffold (συναρτήσεις driver - οδηγοί).

Το παραπάνω σχήμα εξηγεί καλύτερα την διαφορά μεταξύ των stubs και των test drivers. Έστω ότι έχουμε 2 συναρτήσεις(A και B) που θέλουμε να ελέγξουμε, με την A να καλεί την B. Για να ελέγξουμε την A μεμονωμένα πρέπει να αντικαταστήσουμε την B

με μια stub συνάρτηση. Για να ελέγξουμε την Β μεμονωμένα πρέπει να αντικαταστήσουμε την Α με μια driver (οδηγό) συνάρτηση.

Χρησιμοποιώντας τις συναρτήσεις από το κεφάλαιο 6 μπορούμε να δείξουμε την χρησιμότητα των stubs. Έστω ότι θέλουμε να ελέγξουμε την παρακάτω συνάρτηση ενώ δεν έχουμε υλοποιήσει ακόμα τις συναρτήσεις του TStoixeioyOyras.

```
int count_results(TStoixeioyOyras  tmp_lottery, TStoixeioyOyras tmp_result) {
    int * temp1, * temp2, i = 0,j = 0, count = 0, cust = 0;

    TSoyra_getNumbers(tmp_lottery,&temp1);
    TSoyra_getNumbers(tmp_result,&temp2);

    for(i = 0; i < 6; i++) {
        for(j = 0; j < 6; j++) {
            if(temp1[i] == temp2[j]) {
                count++;
                break;
            }
        }
    }
    if(debug){
    TSoyra_getCustomer(tmp_lottery,&cust);
    printf("\nDebug customer %d has %d wins ",cust,count);
    }

    return count;
}
```

Θα χρειαστεί να τις αντικαταστήσουμε λοιπόν με τις παρακάτω συναρτήσεις:

```
void stub_TSoyra_getNumbers(TStoixeioyOyras oura, int * numbers) {
    int i;

    if(numbers!= NULL) {
        for(i = 0; i < 6; i++) {
            numbers[i] = i;
        }
    }
    return;
}
```

και

```
void stub_TSoyra_getCustomer(TStoixeioyOyras oura, int * cust) {
    if(cust != NULL) {
        cust = 1;
    }
    return;
}
```

Έτσι η αρχική μας συνάρτηση μετασχηματίζετε στην παρακάτω :

```
int count_results(TStoixeioyOyras  tmp_lottery, TStoixeioyOyras tmp_result) {
    int * temp1, * temp2, i = 0, j = 0, count = 0, cust = 0;

    stub_ TSoyra_getNumbers(tmp_lottery,&temp1);
```

```
    stub_TSoyra_getNumbers(tmp_result,&temp2);

    for( i = 0; i < 6; i++) {
        for( j = 0; j < 6; j++) {
            if(temp1[i] == temp2[j]) {
                count++;
                break;
            }
        }
    }
    if(debug){
        stub_TSoyra_getCustomer(tmp_lottery,&cust);
        printf("\nDebug customer %d has %d wins ",cust,count);
    }

    return count;
}
```

Έχοντας εξασφαλίσει λοιπόν ότι οι stub συναρτήσεις μας επιστρέφουν πάντοτε έγκυρες τιμές μπορούμε να προχωρήσουμε και να γράψουμε ένα unit test που θα μας ελέγξει την συνάρτηση count_results χωρίς να εξαρτόμαστε από την υλοποίηση των συναρτήσεων του TStoixeioyOyras.

5.3.5.3 Έλεγχος παραδεχόμενων (Acceptance Testing)

Ένα άλλο κομμάτι των ελέγχων που μπορεί να αυτοματοποιηθεί είναι τα acceptance tests. Η διαφορά με το integration testing είναι ότι τώρα οι είναι έλεγχοι που εκτελούνται καθορίζονται από τους ίδιους τους χρήστες του προγράμματος και όχι τους κατασκευαστές του. Αυτού του είδους τα tests μπορεί να διαφέρουν ριζικά σύμφωνα με την εμπειρία του κάθε χρήστη και το χρόνο που έχει διαθέσιμο για αυτούς τους ελέγχους. Μερικές φορές, υπάρχει επίσημο συμβόλαιο ανάπτυξης του συστήματος μεταξύ του πελάτη και της εταιρείας που το υλοποιεί. Ένα τέτοιο συμβόλαιο μπορεί να ορίζει ότι το σύστημα είναι έτοιμο για παράδοση εφόσον περάσει επιτυχώς ένα συγκεκριμένο acceptance test. Άλλες φορές μπορεί να αναφέρεται στο συμβόλαιο ότι κάποια άλλη εταιρεία θα αναλάβει το acceptance test του συστήματος.

Εργαλεία όπως το Selenium προφέρουν στους χρήστες την δυνατότητα να αποθηκεύουν συγκεκριμένες ενέργειες πάνω στο γραφικό περιβάλλον μιας εφαρμογής, όπως η εισαγωγή τιμών σε φόρμα εσόδου ή η περιήγηση σε ένα μενού επιλογών. Στη συνέχεια αναπαράγουν αυτές τις ενέργειες πολλαπλές φορές με διαφορετικές παραμέτρους και μπορούν να ελέγχουν αν τα αποτελέσματα ήταν τα αναμενόμενα.

5.4 Debugging

Στην αρχή του κεφαλαίου αναφερθήκαμε σε μια απλουστευμένη εξήγηση της έννοιας της αποσφαλμάτωσης ενός προγράμματος και εδώ θα αναπτύξουμε λίγο περισσότερο αυτή την έννοια χωρίς να προβούμε όμως σε πολλές τεχνικές λεπτομέρειες και αναλύσεις.

Ένας κύριος λόγος για την ύπαρξη σφαλμάτων σε ένα πρόγραμμα βρίσκεται στο ότι η πολυπλοκότητα ενός προγράμματος είναι συνδεδεμένη με το πλήθος των τρόπων που μπορούν να αλληλεπιδράσουν τα επιμέρους στοιχεία του. Πολλές τεχνικές προγραμματισμού προσπαθούν να περιορίσουν τις συνδέσεις μεταξύ των στοιχείων

του προγράμματος, όπως για παράδειγμα με απόκρυψη πληροφοριών, αφαιρετικότητα και με τη χρήση διεπαφών και των πρόσθετων χαρακτηριστικών που υποστηρίζει η γλώσσα. Υπάρχουν επίσης τεχνικές που εξασφαλίζουν την ακεραιότητα ενός προγράμματος, τη μοντελοποίηση, την ανάλυση των απαιτήσεων αλλά δεν έχουν επιφέρει δραστικές αλλαγές στον τρόπο που κατασκευάζονται τα προγράμματα. Η πραγματικότητα είναι ότι θα υπάρχουν πάντοτε λάθη που θα εντοπίζουμε με τον έλεγχο και θα εξαλείφουμε με την αποσφαλμάτωση.

Πίνακας 2: Παραδειγμα gdb

```
$gdb crash
# Gdb prints summary information and then the (gdb) prompt

(gdb) r
Program received signal SIGFPE, Arithmetic exception.
0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
21        return a / b;

# 'r' runs the program inside the debugger
# In this case the program crashed and gdb prints out some
# relevant information.  In particular, it crashed trying
# to execute line 21 of crash.cc.  The function parameters
# 'a' and 'b' had values 3 and 0 respectively.

(gdb) l
# l is short for 'list'.  Useful for seeing the context of
# the crash, lists code lines near around 21 of crash.cc
```

Η αποσφαλμάτωση είναι μια δύσκολη και χρονοβόρα διαδικασία οπότε πρέπει να προσπαθούμε να την αποφεύγουμε όσο είναι δυνατόν. Τεχνικές που βοηθούν να περιορίζουμε αυτό το χρόνο περιλαμβάνουν τον καλό σχεδιασμό, καλή μορφοποίηση, ελέγχους για οριακές συνθήκες, assertions και ελέγχους ακεραιότητας του κώδικα.

Όταν εντοπίσουμε ένας σφάλμα πρέπει να μελετήσουμε σοβαρά τα διάφορα στοιχεία που έχουμε. Πως για παράδειγμα θα μπορούσε να εμφανιστεί, αν είναι κάτι γνώριμο, αν προκλήθηκε μετά από κάποια πρόσφατη αλλαγή και άλλα. Αν δεν έχουμε ξεκάθαρες ενδείξεις θα πρέπει να προσπαθήσουμε συστηματικά να προσδιορίσουμε την τοποθεσία που συμβαίνει. Ένας τρόπος θα ήταν να περιορίσουμε τα δεδομένα εισόδου μέχρι να εντοπίσουμε τι μικρότερη τιμή που αποτυγχάνει. Ένας άλλος τρόπος είναι να περιορίσουμε στον κώδικα τις περιοχές που δεν συσχετίζονται άμεσα. Εξίσου καλό θα ήταν να προσθέσουμε τμήματα κώδικά που ενεργοποιούνται μόνο όταν ολοκληρωθούν κάποια βήματα στο πρόγραμμα.

Μια άλλη μέθοδος που βοηθάει στο debugging είναι να μπορέσουμε να εξασφαλίσουμε τις συνθήκες κάτω από τις οποίες θα εμφανίζετε πάντοτε το πρόβλημα. Δηλαδή για παράδειγμα όταν χρησιμοποιούμε μια συγκεκριμένη είσοδο ή όταν εκτελούμε κάποια βήματα με συγκεκριμένη σειρά. Με αυτό τον τρόπο μπορούμε να προσδιορίσουμε πιο εύκολα τα τμήματα του κώδικά που επηρεάζει. Έτσι προσθέτοντας επιπλέον γραμμές ελέγχου θα μπορέσουμε να καταλήξουμε στο προβληματικό τμήμα.

Τέλος μια άλλη χρήσιμη τακτική είναι να προσπαθήσουμε να εξηγήσουμε την λειτουργία του προγράμματος μας σε έναν άλλο πιο έμπειρο προγραμματιστή. Η διαδικασία του να προσπαθήσουμε να συμπυκνώσουμε την όλη λογική του κώδικα μας και να την περιγράψουμε με τέτοιο τρόπο που να μπορεί να γίνει εύκολα κατανοητά, βοηθά και

εμάς να κατανοήσουμε βαθύτερα το πρόγραμμα μας. Με αυτό τον τρόπο απλά προβλήματα στη λογική που μας είχαν ξεφύγει πριν, γίνονται πιο εύκολα αντιληπτά.

6. ΜΕΘΟΔΟΙ ΣΧΕΔΙΑΣΜΟΥ ΠΡΟΓΡΑΜΜΑΤΩΝ

Το κεφάλαιο αυτό αναφέρεται σε τεχνικές διαχείρισης μεγάλων και πολύπλοκων προγραμμάτων αποτελούμενων από πολλές ενότητες (modules). Γενικά μία τεχνική υλοποίησης που δουλεύει καλά για ένα πρόγραμμα 100 γραμμών μπορεί να έχει φτωχές επιδόσεις για ένα των 1000 γραμμών και ίσως ακόμη φτωχότερες για ένα πρόγραμμα 10000 γραμμών.

Η μετάβαση από ένα πρόγραμμα λίγων γραμμών σε ένα μεγάλο αρκετών εκατοντάδων δημιουργεί πάντα προβλήματα. Οι προγραμματιστές που επιχειρούν μία τέτοια μετάβαση χωρίς την χρήση οργανογράμματος και σχεδίου ενώ έχουν σαν μόνα εφόδια την εμπειρία τους από μικρά προγράμματα συνήθως αποτυγχάνουν στην πρώτη τους προσπάθεια. Τα κυριότερα προβλήματα που αντιμετωπίζουν είναι τα εξής:

1. Σε ένα μεγάλο πρόγραμμα είναι δύσκολο να προβλεφθούν οι συνέπειες μιας απόφασης που πάρθηκε κατά το αρχικό στάδιο υλοποίησης του. Τα προγράμματα είναι συνήθως τόσο μεγάλα που οι συνέπειες λανθασμένων αποφάσεων φαίνονται πολύ αργότερα (ίσως και μήνες). Ακόμη όμως και όταν αυτές φανούν, είναι πολύ δύσκολο να αναιρέσεις τις συνέπειες της αρχικής απόφασης με αποτέλεσμα να αποδεχόμαστε τις λάθος επιλογές.
2. Οι διαδικασίες, της αποσφαλμάτωσης (debugging) καθώς και του ελέγχου (testing) γίνονται περίπλοκο θέμα.
3. Πνίγονται στην λεπτομέρεια. Υπάρχουν τόσα πολλά πράγματα να κάνουν και να θυμηθούν, που ξεχνούν να εκτελέσουν βασικές λειτουργίες, να υλοποιήσουν κομμάτια του κώδικα που θεωρούνται κρίσιμα, να εκτελέσουν ελέγχους και άλλα. Έτσι χάνουν τον έλεγχο για το τι έχει γίνει μέχρι τώρα και για το τι χρειάζεται να γίνει ακόμη.

Για τους λόγους αυτούς θα πρέπει να αναπτύξουμε και να χρησιμοποιήσουμε οργανωτικές δομές που βοηθούν την σωστή ανάπτυξη μεγάλων και πολύπλοκων προβλημάτων.

6.1 Δομημένη Ανάπτυξη Προγραμμάτων

Η Δομημένη ανάπτυξη προγραμμάτων είναι μία τεχνική υλοποίησης προγραμμάτων που συστηματοποιεί και οργανώνει όλο τον κύκλο σχεδίασης, κωδικοποίησης και ελέγχου των προγραμμάτων. Ο απώτερος σκοπός της τεχνικής αυτής είναι η ανάπτυξη σωστών και αξιόπιστων προγραμμάτων αποφεύγοντας τα λάθη και βοηθώντας τον έλεγχο για τον εντοπισμό τους. Η Δομημένη ανάπτυξη προγραμμάτων επιχειρεί την ανάπτυξη λογισμικού που θα ελαχιστοποιεί τα έξοδα προσωπικού για την δημιουργία του (εργατοώρες) αυξάνοντας ταυτόχρονα την παραγωγικότητα.

Η μεθοδολογία δομημένης ανάπτυξης προγραμμάτων αποτελείται από τρία συστατικά μέρη:

1. Φιλοσοφία Σχεδιασμού από το γενικό προς το ειδικό (Top-down design)
2. Ανεξαρτησία ενοτήτων προγραμμάτων
3. Αρχές δομημένου προγραμματισμού.

Σε αυτό το κεφάλαιο θα ασχοληθούμε με το πρώτο συστατικό δηλαδή τον σχεδιασμό top - down.

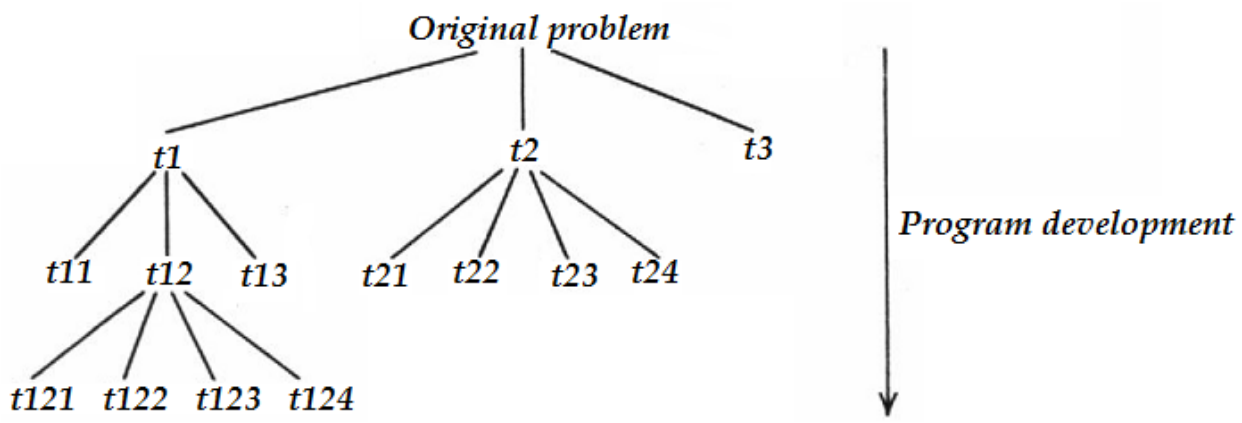
6.2 Top - Down σχεδιασμός προγράμματος

Οι τεχνικές του σχεδιασμού top-down σας είναι ήδη γνωστές από την επαφή σας με άλλα θέματα της καθημερινής πραγματικότητας Η ανάπτυξη του σκελετού μίας έκθεσης ακολουθεί αυτή την τεχνική με την όλο και πιο λεπτομερή προσέγγιση του θέματος το οποίο αναπτύσσεται.

6.2.1 Διαδικασία Εκλέπτυνσης

Στα προγράμματα η μέθοδος σχεδιασμού top-down αρχίζει με τον γενικό σκοπό του προγράμματος. Δηλαδή με το τί θέλουμε να κάνει το πρόγραμμα και όχι πως θα το πετύχουμε. Καθορίζουμε μία πολύ γενική λύση που κατορθώνει να πετύχει τους σκοπούς αυτούς καλώντας μία σειρά διαδοχικών βημάτων-οδηγιών. Κάθε διαδοχικό βήμα αναπτύσσει βαθύτερα τον στόχο και με τον τρόπο αυτό πετυχαίνουμε ολοένα και μεγαλύτερο εκλεπτυσμό των στόχων (Stepwise Refinement).

Όσο η διαδικασία εκλέπτυνσης συνεχίζεται, περιγράφονται με μεγαλύτερη λεπτομέρεια οι στόχοι χαμηλού επιπέδου που απαιτούνται για να υλοποιηθεί η λειτουργία υψηλού επιπέδου που έχει περιγραφεί. Η διαδικασία αυτή περιγράφεται με μία δενδροειδή δομή (treelike structure) όπως φαίνεται στο σχήμα 9.



Σχήμα 9: Δέντρο ανάπτυξης προγράμματος.

Στο σχήμα οι υποστόχοι (subtasks) t1,t2 και t3 απαιτούνται για να λύσουν το αρχικό πρόβλημα. Οι υπόστοχοι t11,t12 και t13 απαιτούνται για την λύση του υποστόχου t1 και ούτως καθεξής.

Η διαδικασία της εκλέπτυνσης σταματάει όταν ένας υπόστοχος (όπως το t121 του σχήματος 9) είναι αρκετά απλό έτσι ώστε να μπορεί να υλοποιηθεί κατευθείαν χωρίς να χρειάζεται να θέσουμε κάποιους επιπρόσθετους υποστόχους χαμηλότερου επιπέδου.

Κάθε υπόστοχος ελέγχεται ότι είναι σωστός πριν εκλεπτυνθεί περισσότερο. Έτσι ο στόχος t1 θα ελεγχθεί τελείως για να δείξουμε και τυπικά ότι είναι σωστή η χρήση του. Στην συνέχεια για ακολουθήσει ο σχεδιασμός και η κωδικοποίηση των υπόστοχων t11, t12 και t13.

Για παράδειγμα, θα εφαρμόσουμε την φιλοσοφία σχεδίασης top-down για να υλοποιήσουμε ένα πρόγραμμα κληρώσεων και υπολογισμού των κερδών σε μια λοταρία. Για χάριν ευκολίας υποθέτουμε ότι οι πελάτες έχουν καταχωρίσει από πριν τις επιλογές τους για τις επόμενες κληρώσεις. Όποτε το πρόγραμμα μας θα δέχεται ένα αρχείο με τις επιλογές των χρηστών και θα προσομοιώνει την κάθε κλήρωση και θα τυπώνει τα αποτελέσματα ανά κλήρωση και πελάτη.

6.2.2 Αρχικά βήματα για το σχεδιασμό της λύσης

Στον προγραμματισμό top-down με εκλεπτυσμένα βήματα, αρχίζουμε από το ανώτατο εννοιολογικό επίπεδο, με το να φανταζόμαστε μία γενική, αφαιρετική λύση. Αυτό μπορεί να εκφραστεί με τη σχεδίαση μίας προγραμματιστικής στρατηγικής σε γενικές γραμμές πριν την επιλογή κάποιας αναπαράστασης χαμηλού επιπέδου δεδομένων ή αλγορίθμων. Στη συνέχεια χρησιμοποιώντας βήμα προς βήμα προοδευτική

εκλέπτυνση, κάνουμε επιλογές για να την εμπλουτίσουμε με λεπτομέρειες, όπως στη διαδικασία γραψίματος μίας ιστορίας που αρχίζουμε με μία περιγραφή της πλοκής της. Γενικεύοντας, επιλέγουμε αλγόριθμους και δομές δεδομένων σε συνδυασμό για να υλοποιήσουμε τα τμήματα της λύσης που σχεδιάστηκε προηγουμένως σε υψηλού επιπέδου περιγραφές. Τελικά, φτάνουμε σε ένα εκτελέσιμο πρόγραμμα σε C.

Συνήθως υπάρχουν πολλοί τρόποι να λυθεί ένα πρόβλημα. Είναι καλή ιδέα να μην σταματάτε να ψάχνετε για λύσεις μόλις μία αρχική λύση έχει σχεδιαστεί. Αντίθετα, μπορεί να αξίζει τον κόπο να προσπαθήσετε να φανταστείτε τρεις ή περισσότερες λύσεις, και μετά να συγκρίνετε τα πλεονεκτήματα και τα μειονεκτήματα τους για να πραγματοποιήσετε μία πλήρη εργασία για το ψάξιμο μίας καλής λύσης.

Μία πιθανή λύση ακολουθεί την παρακάτω δομή:
Πρόγραμμα προσομοίωσης κληρώσεων ανά επίπεδα:

1) Ανάγνωση και επεξεργασία δεδομένων εισόδου.
   1-1) Ανάγνωση δεδομένων από αρχείο.
   1-2) Αποθήκευση και επεξεργασία δεδομένων ανά πελάτη και επιθυμητή κλήρωση.
      1-2-1) Δυναμική δημιουργία απαραίτητων δομών δεδομένων.
      1-2-2) Εισαγωγή και ταξινόμηση δεδομένων.

2) Προσημείωση κληρώσεων.
   2-1) Υλοποίηση μηχανισμού κλήρωσης.
      2-1-1) Γεννήτρια τυχαίων αριθμών.
   2-2) Επαναχρησιμοποίηση του μηχανισμού για ν κληρώσεις και οργάνωση αποτελεσμάτων.
      2 - 2 - 1) Δυναμική δημιουργία απαραίτητων δομών δεδομένων.
      2 - 2 - 2) Εκτέλεση ν κληρώσεων.

3) Υπολογισμός αποτελεσμάτων και εκτύπωση.
   3 - 1) Υπολογισμός κερδών ανά πελάτη και κλήρωση.
   3 - 2) Εκτύπωση αποτελεσμάτων.

6.2.3 Στόχοι ανωτέρου επιπέδου

Ας διευκρινίσουμε τη διαδικασία του προγραμματισμού top-down με εκλεπτυσμένα βήματα περνώντας μέσω των εκλεπτυσμένων βημάτων για τη παραπάνω λύση. Μπορούμε να αρχίσουμε φτιάχνοντας μία προγραμματιστική στρατηγική σε C περιγράφοντας τους στόχους του ανώτερου επιπέδου.

```
int main (int argc, char *argv[ ])
{
    Βήμα 1) Ανάγνωση και επεξεργασία δεδομένων εισόδου
    Βήμα 2) Προσημείωση κληρώσεων
    Βήμα 3) Υπολογισμός αποτελεσμάτων και εκτύπωση
}
```

Για να ληφθούν οι εισαγόμενοι αριθμοί από τους χρήστες ανά κλήρωση, θα αποφασίσουμε να χρησιμοποιήσουμε έναν αριθμό από δομές τύπου ουράς που θα προσαρμόζουμε ανάλογα με το πλήθος των δεδομένων. Μπορούμε τότε να γράψουμε ένα απλό πρόγραμμα εισόδου που διαβάζει από το αρχείου εισόδου τις επιλογές του κάθε πελάτη για κάθε κλήρωση και να τις αποθηκεύει σε αυτές τις δομές. Αυτό είναι το πρώτο μας εκλεπτυσμένο βήμα, στο οποίο επιλέγουμε την αναπαράσταση των δεδομένων και έναν αλγόριθμο στον ίδιο χρόνο.

Μετά τη δήλωση των δεδομένων, μπορούμε να γράψουμε ένα απλό, αλγόριθμο για να δέχεται ως είσοδο το όνομα ενός αρχείου με γραμμές από αριθμούς σε

προκαθορισμένες θέσεις. Επιλέξαμε το ακόλουθο πρότυπο για το αρχείο μας : «αριθμός πελάτη» «αριθμός κλήρωσης» «επιλογή 1» … «επιλογή 6» «ποσό στοιχήματος»

Ενδεικτικά δεδομένα για παράδειγμα :

1 1 12 6 54 9 47 21 1.5

1 2 12 6 54 9 47 21 1.523

2 1 2 16 4 35 17 26 34

3 2 22 33 11 50 4 27 2.2

4 3 30 20 16 9 7 1 11

Ποιο λεπτομερείς εκδόσεις μπορούν να προσφέρουν προστασία έναντι σε πιθανά λάθη του χρήστη, όπως το να μη δίνει έξι ποσά, ή να πληκτρολογεί λανθασμένα μη ακέραιο αριθμό κλήρωσης.

6.2.4 Εκλέπτυνση του πρώτου βήματος

Στη συνέχεια, χρειάζεται να κάνουμε βηματική εκλέπτυνση στοχεύοντας στο ανώτερο επίπεδο ξεκινώντας από το Βήμα 1) Ανάγνωση και επεξεργασία δεδομένων εισόδου. Μπορούμε να προσπαθήσουμε να επιλέξουμε τον τύπο δεδομένων μας πριν να σχεδιάσουμε την στρατηγική για τον αλγόριθμο, ή, εναλλακτικά, μπορούμε να προσπαθήσουμε να σχεδιάσουμε τον αλγόριθμο πριν να επιλέξουμε τις λεπτομέρειες της δομής των δεδομένων. Η δεύτερη εναλλακτική ονομάζεται αναβολή της επιλογής της αναπαράστασης των δεδομένων.

Ας προσπαθήσουμε αυτή τη δεύτερη ιδέα, σχεδιάζοντας τον αλγόριθμο πριν την επιλογή της δομής δεδομένων. Για να το κάνουμε αυτό, επιλέξαμε να χρησιμοποιήσουμε τις λειτουργίες μιας αφηρημένης δομής ουράς που δεν δημιουργεί δεσμεύσεις στην αναπαράσταση.

```
lottery_data * read_data(char * filename)  {
    FILE *fp;
    lottery_data * draw_data;
    QueueElement *temp_element;
    char buf[256];
    int i, overflow = 0;

    /* Dynamic allocation of memory */
    draw_data = (lottery_data *)calloc(1,sizeof(lottery_data) );
    draw_data->lotteries = (QueueHandle * ) calloc
                          (MAXDRAWSIZE,sizeof(QueueHandle *));
    for(i = 0; i < MAXDRAWSIZE; i++) {
        draw_data->lotteries[i]=QueueConstructor();
    }
    draw_data->count=0;

    /* Open data file */
    if( (fp=fopen(filename,"r")) == NULL ) {
        printf("\nFile %s not found\n",filename);
        exit(1);
    }
    printf("\nReading from file %s",filename);
    fflush(stdout);
```

```
    /* process data file */
    while( fgets(buf,sizeof(buf),fp) != NULL ) {
        temp_element = parse_line(buf);
        printf("\nAdding an new gamesubscription");
        fflush(stdout);
        /* add a new record */
        InsertQueue(draw_data->lotteries[temp_element->lottery_id],
        temp_element, &overflow);
        }

        printf("\nFinished reading from file");
        fclose(fp);

        return draw_data;
}
```

6.2.4.1 Εναλλακτική υλοποίηση συναρτήσεων με χρηση stubs

Σε αυτή τη προγραμματιστική στρατηγική καλείται η συνάρτηση **parse_line(buf)** που δεν έχουμε υλοποιήσει. Όπως και επίσης και η συνάρτηση InsertQueue() που ακόμα δεν έχουμε καταλήξει σε ποια υλοποίηση θα αντιστοιχεί. Όταν καλούμε συναρτήσεις που δεν έχουμε γράψει ακόμα, μπορούμε να βάλουμε τα ονόματα τους σε μια λίστα υπολειπόμενων συναρτήσεων για γράψιμο. Όταν έχουμε γράψει όλες τις συναρτήσεις της λίστας, έχουμε ολοκληρώσει ένα επίπεδο της διαδικασίας της βηματικής εκλέπτυνσης. Αυτή η μέθοδος καλείται top-down μέθοδος, γιατί η δραστηριότητα της δημιουργίας προγραμματιστικών στρατηγικών στο ανώτερο επίπεδο οδηγεί στη δημιουργία κλήσεων συναρτήσεων, των οποίων οι χαμηλού επιπέδου ορισμοί δεν έχουν γραφεί ακόμα. Συνεπώς καλούμαστε να συμπληρώνουμε τα επίπεδα και τις λεπτομέρειες της γενικής προγραμματιστικής στρατηγικής με σειρά top to bottom προχωρώντας ανά στρώση. Αρχίζουμε με την ανώτατη (ή πιο αφαιρετική) στρωμάτωση, και προοδευτικά καθορίζουμε τα χαμηλότερα (ή τα πιο λεπτομερή) στρώματα.

Ένας εύκολος τρόπος για να βελτιώσουμε τον έλεγχο του προγράμματος μας πριν ολοκληρώσουμε το γράψιμο του είναι να χρησιμοποιήσουμε συναρτήσεις **stubs** για να αντικαταστήσουμε προσωρινά την υλοποίηση των συναρτήσεων που δεν έχουμε γράψει ακόμα. Με αυτό τον τρόπο θα μπορέσουμε να μεταγλωττίσουμε το πλήρες πρόγραμμα μας και έτσι να εντοπίσουμε πρόωρα τυχόν συντακτικά και λογικά λάθη. Επίσης θα μας βοηθήσει και στην σωστή εκτέλεση του προγράμματος. Μετά από μία ορθή εκτέλεση είμαστε σίγουροι πως το επίπεδο αυτό έχει οριστεί σωστά έτσι ώστε να προχωρήσουμε την ανάλυση μας σε εκλέπτυνση άλλων υποστόχων.

Μπορούμε να χρησιμοποιήσουμε 2 διαφορετικές προσεγγίσεις για stub συναρτήσεις. Μπορούμε αρχικά να υλοποιήσουμε μια συνάρτηση στην πιο απλή μορφή της που απλώς να τυπώνει ένα μήνυμα. Αυτό θα πιστοποιεί πως ο έλεγχος του προγράμματος έχει φθάσει στην αντίστοιχη ρουτίνα. Επίσης θα εξασφαλίζει πως η σύνδεση της ρουτίνας με το πρόγραμμα είναι η ενδεδειγμένη. Για παράδειγμα στην παρακάτω υλοποίηση απλώς τυπώνουμε ένα μήνυμα και επιστέφουμε null.

```
QueueElement * parse_line(char * line) {
    /* print a message */
    printf("Parse line function successfully called");

    return null;
```

```
}
```

Μια διαφορετική υλοποίηση που θα μπορούσε ενδεχομένως να μας φανεί πιο χρήσιμη θα ήταν η υλοποίηση μιας stub συνάρτησης με πραγματικές επιστρεφόμενες τιμές. Μια συνάρτηση δηλαδή που θα μας βοηθήσει και στον έλεγχο της λειτουργικότητας του υπόλοιπου προγράμματος. Για να το επιτύχουμε αυτό αρκεί να καθορίσουμε τις κατάλληλες αρχικές τιμές.

```
QueueElement * parse_line(char * line) {

    QueueElement * temp;
    int i = 0;

    /* Allocate memory for the data object */
    temp=(QueueElement *)calloc(1,sizeof(QueueElement ));
    temp->customer_id = 1;
    temp->lottery_id =1;
    for(i = 0; i < 6; i++) {
    temp-> numbers [i]= i;
    }
    temp->wins=0;
    temp->amount= 0;

    return temp;
}
```

Χρησιμοποιώντας την ίδια λογική θα μπορούσαμε να υλοποιήσουμε stub συναρτήσεις και για την InsertQueue. Στην απλή μορφή της απλώς εκτυπώνουμε ένα μήνυμα:

```
void InsertQueue(const QueueHandle QueuePtr, QueueElement * const ElementPtr,
                 int *overflow){
    printf("Queue element inserted successfully");
}
```

Εναλλακτικά μπορούμε να υλοποιήσουμε αυτή τη συνάρτηση ώστε να αναθέτει πάντοτε το εισαγόμενο στοιχείο στην πρώτη θέση της ουράς. Με αυτό τον τρόπο θα έχουμε μια υποτυπώδη ουρά με ένα στοιχείο για να χρησιμοποιήσουμε μετέπειτα στο υπόλοιπο πρόγραμμα μας .

```
void InsertQueue(const QueueHandle QueuePtr, QueueElement * const ElementPtr,
                 int *overflow){
    *overflow = 0;
    QueuePtr->metritis=1;
    QueuePtr->pinakas[0] = ElementPtr;
    QueuePtr->piso = 0;
}
```

6.2.4.2 Υλοποίηση των υπόλοιπων συναρτήσεων του πρώτου βήματος

Στον παρακάτω κώδικα θα παρέχουμε την αφαιρετική προγραμματιστική στρατηγική για την επεξεργασία μιας γραμμής του αρχείου εισόδου μας και αποθήκευσης των αριθμών σε ένα στοιχειό της δομής μας.

```
/* Pre: data line needed as input
 After: Line is parsed and a pointer to the data object is returned
*/
```

```
QueueElement * parse_line(char * line)
{
    /* Initialize local variables */
    char * args[256];
    int I = 0, count = 0, nums[8];
    float amount;

    QueueElement * temp;

    args[0] = line;
    /* process line , one character at a time */
    while (*line != '\0') {
        while ((*line == ' ') || (*line == '\t') || (*line == '\n')) {
            /* replace empty characters with \0 */
            *line++ = '\0'
        }
        args[count] = line; /*  save the arg */
        count++;
        while ((*line != '\0') && (*line != ' ') && (*line != '\t') && (*line != '\n')) {
            if(*line == '#') {
                break;
            }
            else {
                line++;
            }
     }
   }
    /* Allocate memory for the data object */
    temp=(QueueElement *)calloc(1,sizeof(QueueElement ));
    for(i = 0; i < 8; i++){
        nums[i]=atoi(args[i]);
    }

    amount=atof(args[8]);
    /* Store data */
    StoreElement(temp,nums,amount);

    return temp;
}
```

Σημειώστε ότι έχουμε δώσει όλες τις εκλεπτύνσεις όλων των βημάτων στην προγραμματιστική στρατηγική του ανώτερου επιπέδου (Βήμα 1), αλλά δεν έχουμε επιλέξει ακόμα αναπαράσταση δεδομένων για την ουρά που θα χρησιμοποιηθεί για να αποθηκεύσει τους αριθμούς, τα ποσά και τις επαναλήψεις. Έστω ότι σε αυτό το σημείο, γνωρίζουμε όλες τις αφαιρετικές λειτουργίες της ουράς που θα χρειαστούν για τη λύση του προβλήματος. Υπάρχουν δύο τέτοιες λειτουργίες: η εισαγωγή των επιλογών ενός πελάτη και των αποτελεσμάτων μιας κλήρωσης και η εξαγωγή επιλεκτικών στοιχείων για τους υπολογισμούς μας. Μόλις μάθουμε τη φύση των λειτουργιών εισαγωγής και εξαγωγής που θα χρησιμοποιηθούν, θα είμαστε σε καλή θέση για να κρίνουμε τι είδη αναπαραστάσεων και αλγορίθμων θα δουλεύουν καλά.

Σημειώστε επίσης ότι έχουμε δώσει τις προγραμματιστικές στρατηγικές για τη δημιουργία της ουράς (QueueConstructor), για την εισαγωγή νέων δεδομένων στην

ουρά (StoreElement και InsertQueue), χωρίς ακόμα να έχουμε επιλέξει την αναπαράσταση των δεδομένων για την ουρά. Αυτό σημαίνει ότι έχουμε χρησιμοποιήσει αφαιρετικότητα για να ματαιώσουμε τη δέσμευση στην επιλογή της αναπαράστασης των δεδομένων για τις ουρές. Συνεπώς, είμαστε ελεύθεροι να σκεφτούμε διαφορετικές αναπαραστάσεις για την ουρά (όπως υλοποίηση ουράς με πίνακα, με στοίβα, με απλή συνδεδεμένη λίστα και τα λοιπά). Ανάλογα με την επιλογή μας, θα έχουμε στη διάθεση μας διάφορες υλοποιήσεις για τον ΑΤΔ ουρά από έτοιμα γραμμένα δομοστοιχεία.

6.2.4.3 Επιλέγοντας Αναπαράσταση Δεδομένων για την ουρά

Δε θα εμπλακούμε εδώ στα συγκρίσιμα πλεονεκτήματα και μειονεκτήματα των πολλών επιλογών αναπαραστάσεων για τις ουρές. Ωστόσο, δίνουμε έμφαση στο ότι αυτή είναι η κατάλληλη στιγμή για την επιλογή πλεονεκτικής αναπαράστασης δεδομένων. Η επιλογή θα είναι ιδιαίτερα σημαντική αν χρειάζεται να χρησιμοποιήσουμε πολλές και μεγάλες ουρές με αποδοτική εισαγωγή και χρόνους αναζήτησης για πολλά αντικείμενα. Αυτό θα ήταν σημαντικό, για παράδειγμα, αν μας δινόταν το πρόβλημα της επεξεργασίας τεράστιου όγκου πελατών και κληρώσεων. Στην παρούσα περίπτωση, όμως, τα δεδομένα μας είναι τόσο μικρά που σχεδόν κάθε επιλογή λογικής αναπαράστασης που κάνουμε για της ουρά θα είναι αποδοτική. Έτσι θα προβούμε στην επιλογή μια υλοποίησης ουράς με πίνακες σταθερού μεγέθους και μιας ουράς με δυναμικούς πίνακες ανάλογα με τις ανάγκες μας.

Οι παρακάτω προσδιοριστικοί τύποι και δηλώσεις μεταβλητών καθορίζουν τις δομές δεδομένων που θα χρησιμοποιηθούν σε αυτή την αναπαράσταση δεδομένων:

Ο τύπος στοιχείου της ουράς μας είναι:

```
typedef struct {
    int customer_id;   /* unique customer identifier */
    int lottery_id;    /* unique draw identifier */
    int numbers[6];    /* customer picks */
    int wins;          /* total winnings */
    float amount;      /* stake amount */
} QueueElement;
```

Θα χρησιμοποιήσουμε τον ίδιο τύπο για τις επιλογές των πελατών και τις κληρώσεις.

Για τις επιλογές των πελατών έχουμε την παρακάτω δομή:

```
struct  LotterySelections {
    QueueHandle * lotteries;
    int count;
};
```

Ενώ για τα αποτελέσματα των κληρώσεων:

```
struct LotteryResults {
    QueueHandle  lot_draw;
    int count;
};
```

Μόλις πάρουμε την απόφαση να χρησιμοποιήσουμε αυτή την αναπαράσταση, οι λεπτομέρειες του τελικού βήματος εκλέπτυνσης των προγραμματιστικών στρατηγικών για το Βήμα 1 είναι αρκετά εύκολο να εμπλουτιστούν Για παράδειγμα, ας εκλεπτύνουμε την προγραμματιστική στρατηγική σε ένα λεπτομερές, εκτελέσιμο πρόγραμμα. Θα υλοποιήσουμε τη δική μας εκδοχή για τις συναρτήσεις του τύπου στοιχειού της ουράς (QueueElement) που καλούμε απευθείας αλλά και μέσα από την υλοποίηση του τύπου ουράς που έχουμε έτοιμη.

Συνάρτηση αποθήκευσης τιμών σε ένα νέο στοιχείο:

```
/* Pre : Target data object to store the value, data array
 After: Target data object is populated with the provided numbers from the array
*/
void StoreElement(QueueElement *target,int *nums,float amount) {
    int i = 0;

    target->customer_id=nums[0];
    target->lottery_id=nums[1];
    for(i = 0; i < 6; i++)  {
        target->numbers[i]=nums[i+2];
    }
    target->amount=amount;

    return;
}
```

Συνάρτηση αποθήκευσης τιμών από ένα υπάρχον στοιχείο σε ένα νέο:

```
/* Pre : Target data object to store the value,  Source data object
 After: Target  data object is populated with the provided numbers from the source
*/
void ElementSetValue  (QueueElement *target, QueueElement source) {
    int i = 0;

    target->customer_id=source.customer_id;
    target->lottery_id=source.lottery_id;
    for(i = 0; i < 6; i++) {
        target->numbers[i]=source.numbers[i];
    }
    target->amount=source.amount;
    target->wins=0;

    return;
}
```

6.2.5 Εκλέπτυνση του δεύτερου βήματος

Τώρα που γνωρίζουμε τις λεπτομέρειες για την αναπαράσταση δεδομένων για τον πίνακα, μπορούμε να προχωρήσουμε στην υλοποίηση και των άλλων βημάτων της προγραμματιστικής στρατηγικής μας.

Ακολουθεί το Βήμα 2

```
/* Pre: Number of draws required
After : Draw results have been generated */
rdata * make_draws(int draws) {
    rdata * results;
    QueueElement *temp_element;

    struct timeval start;
    /* Initialization  for the random number sequence at the current time */
    gettimeofday(&start, NULL);
    srand(start.tv_usec*1000);
```

```
    int i = 0, overflow;

    /* Dynamic memory allocation for the draws  */
    results=(rdata *)calloc(1,sizeof(rdata) );
    results->lot_draw=QueueConstructor();

    /* Repeating and executing draws */
    for(i = 0; i < draws; i++) {
        temp_element=(QueueElement *)calloc(1,sizeof(QueueElement ));
        draw_numbers(i,temp_element);
        InsertQueue(results->lot_draw,temp_element,&overflow);
        free(temp_element);
    }

    return results;
}
```

Οι στόχοι επιπέδου του Βήματος 2 μπορούν να εκλεπτυνθούν με την εισαγωγή κλήσης συνάρτησης για της προσημείωση της κλήρωσης και της επιλογής τυχαίων αριθμών:

```
/* Pre : target data object required to store the results
After : results are generated in a random fashion and are stored in the data object
*/
void draw_numbers(int draw, QueueElement * temp){
    int num[6], x, j, i, found = 0, ok = 0;

    temp->lottery_id = draw;

    for(i = 0; i < 6;i++) {
        num[i] = 0;
    }
    /* Repeat until you have 6 different random numbers */
    for(i = 0; i < 6; i++) {
        ok = 0;
        while(!ok) {
            /* random number generation */
            x = rand() % 50 + 1;
            found = 0;
            j=0;
            /* check if it was drawn before */
            while(!found && j < i) {
                if(x == num[j]) {
                found = 1;
            }
            j++;
        }
            /* if number was unique store it */
            if(!found){
                ok = 1;
            }
            num[i] = x;
            temp-> numbers[i] = x;
        }
```

```
    }
    return;
}
```

6.2.6 Εκλέπτυνση του τρίτου Βήματος

Για το Βήμα 3 έχουμε την εξής στρατηγική:

```
/* Pre: lottery data required
After : results are displayed
*/
void show_results(lottery_data * draw_data, rdata * results, int draws, char * filename) {
    int i = 0, underflow = 0, overflow = 0, count = 0, cust_id, * nums;
    /* Local variables*/
    QueueElement * tmp_result, * tmp_lottery;
    QueueHandle * customers;
    FILE * fp;

    /* open data file */
    if ((fp = fopen(filename, "w")) == NULL) {
        printf("\nFailed to open file %s .\n", filename);
        exit(1);
    }
    printf("\nCreated file %s to store results", filename);
    /* allocate memory */
    customers = (QueueHandle * ) calloc(MAXCUSTSIZE, sizeof(QueueHandle));
    for (i = 0; i < MAXCUSTSIZE; i++) {
        customers[i] = QueueConstructor();
    }
    /* Process the draws */
    for (i = 0; i < draws; i++) {
        nums = (int * ) calloc(6, sizeof(int));
        tmp_result = (QueueElement * ) calloc(1, sizeof(QueueElement));

        QueuePop(results - > lot_draw, tmp_result, & underflow);
        ElementGetNumbers( * tmp_result, & nums);

        fprintf(fp, "\nDraw %d Results: %d %d %d %d %d %d", i + 1, nums[0], nums[1],
                nums[2], nums[3], nums[4], nums[5]);
        printf("\n Processing results for draw: %d", i + 1);
        while (!IsEmptyQueue(draw_data - > lotteries[i])) {
        /* Pick the first item from the queue */
        tmp_lottery = (QueueElement * ) calloc(1, sizeof(QueueElement));
        QueuePop(draw_data - > lotteries[i], tmp_lottery, & underflow);
        count = count_results( * tmp_lottery, * tmp_result);
        if (count >= 3) {
            ElementUpdateWins(tmp_lottery, count);
        }
        ElementGetCustomer( * tmp_lottery, & cust_id);
        /* Add the resulted customers back to the customer queue */
        printf("\nCalculating winnings for customer:%d", cust_id);
        InsertQueue(customers[cust_id], tmp_lottery, & overflow);
        free(tmp_lottery);
        }
```

```
    free(nums);
    free(tmp_result);
    }

    print_results(customers, fp);

    clear_mem(customers, MAXCUSTSIZE);
    fclose(fp);

    return;
}
```

Στην παραπάνω υλοποίηση παρατηρούμε την κλήση κάποιων συναρτήσεων του τύπου στοιχείου ουράς (ElementGetNumbers, ElementUpdateWins, ElementGetCustomer) που θα χρειαστεί να υλοποιηθούν όπως και επίσης οι κλήσεις των συναρτήσεων του χαμηλότερου υποεπιπέδου για τον υπολογισμό και της εκτύπωση των αποτελεσμάτων (count_results, print_results).

```
/* Pre : draw results and customer picks
After: matching count returned
*/
int count_results(QueueElement  tmp_lottery, QueueElement tmp_result) {
    int * temp1, * temp2,  i = 0, j = 0, count = 0, cust = 0;

    ElementGetNumbers(tmp_lottery,&temp1);
    ElementGetNumbers(tmp_result,&temp2);

    for( i = 0; i < 6; i++) {
        for( j = 0; j < 6; j++) {
            if(temp1[i] ==t emp2[j]) {
                count++;
                break;
            }
        }
    }
    return count;
}

/* Pre : Customer list provided
After : results are printed in the file
*/
void print_results(QueueHandle * customers,FILE * fp) {
    int i = 0, underflow = 0, count = 0;
    float total = 0.0;

    QueueElement * temp;

    for(i =0 ; i < MAXCUSTSIZE; i++) {
        total = 0.0;
        count = 0;
        temp=(QueueElement *)calloc(1,sizeof(QueueElement ));
        while(!IsEmptyQueue(customers[i])) {
            /* remove one queue item each time */
```

```
                QueuePop(customers[i],temp,&underflow);
                /* print the output to the given file pointer */
                fprintf(fp,"\nCustomer:%d Draw:%d Wins:%d Amount:%.2lf",
                        temp->customer_id,temp->lottery_id,temp->wins,
                        (temp->amount*((float)temp->wins/6)*10) );
                total+=temp->amount*((float)temp->wins/6)*10;
                count++;
            }
            if( count > 0) {
                fprintf(fp,"\n=> Customer:%d  Total Amount:%.2lf",
                        temp->customer_id,total );
                free(temp);
            }
        }
        return;
}
```

6.2.7 Τελική μορφή προγράμματος

Τώρα που όλες οι λεπτομερείς, χαμηλού επιπέδου συναρτήσεις και αναπαραστάσεις δεδομένων έχουν ολοκληρωθεί, μπορούμε να τις συναρμολογήσουμε σε ένα τελικό πρόγραμμα.

```
#include <stdio.h>
#include <stdlib.h>
#include "lottery.h"

extern int debug;

int main (int argc, char *argv[])
{
    int draws;

    lottery_data * draw_data = NULL;
    rdata * results = NULL;

    if (argc < 4) {
        printf("\n Usage: %s <data file> <num of draws> <results file>",argv[0]);
        exit(1);
    }

    if ( (draws=atoi(argv[2]))  <= 0 ) {
        printf("Invalid number of draws:%s",argv[2]);
        exit(1);
    }

    /* checks for extra debugging options */
    if(argc == 5) {
        debug=atoi(argv[4]);
        if(debug <= 0) {
            debug = 0;
        }
        else {
```

```
            debug=1;
        }
    }

    /* Step 1 Reading and processing input data */
    printf("\nStep 1 - Read data from file: Started");
    draw_data=read_data(argv[1]);
    if(draw_data==NULL) {
        printf("\nUnexpected error. Exiting...");
        exit(1);
    }
    printf("\nStep 1 : Completed");

    /* Step 2 predetedermined draws */
    printf("\nStep 2 - Simulate lottery draws: Started");

    results=make_draws(draws);
    if(results==NULL) {
        printf("\nUnexpected error. Exiting...");
        exit(1);
    }
    printf("\nStep 2 : Completed");

    /* Step 3 Calculating results and printing */
    printf("\nStep 3 - Print results: Started");
    show_results(draw_data,results,draws,argv[3]);
    printf("\nStep 3 : Completed");

    clear_lot(draw_data);
    clear_res(results);

    return 0;
}
```

Το τελικό πρόγραμμα μαζί με κάποιες επιπλέον λεπτομέρειες υλοποίησης παρατίθεται στο τέλος του αρχείου.

6.3 Πλεονεκτήματα της μεθόδου σχεδίασης top-down.

Το κυριότερο πλεονέκτημα της χρήσης του top-down σχεδιασμού είναι ότι καθιστά εύκολο την διαχείριση ενός προγράμματος.

Ο διανοητικός έλεγχος του προγράμματος επιτυγχάνεται με την ανάπτυξη του προβλήματος από τους πιο γενικούς στόχους στις πιο ειδικές λεπτομέρειες. Ο διανοητικός αυτός έλεγχος επιτυγχάνεται με μία διαδικασία που λέγεται αφαίρεση abstraction). Με την διαδικασία της αφαίρεσης, ασχολούμαστε αρχικά με μία λειτουργία από την γενική της άποψη χωρίς να εισερχόμαστε σε λεπτομέρειες. Μπορούμε έτσι να αναπτύξουμε μεγάλα και πολύπλοκα προβλήματα, απορρίπτοντας καταρχάς όλες τις ασήμαντες λεπτομέρειες που αφορούν τα χαμηλά επίπεδα και επικεντρώνοντας την προσοχή μας σε μερικά μόνο σημεία που αφορούν δομές υψηλού επιπέδου.

Ένα άλλο σημαντικό σημείο είναι πως πρέπει να έχουμε περισσότερο υπόψη μας τον αριθμό των στόχων και όχι τον τρόπο υλοποίησης τους. Με τον τρόπο αυτό οι εκλεπτύνσεις ενός προβλήματος γίνονται ακόμη πιο κατανοητές, αφού αυτές πάντα εκφράζονται με ένα σχετικά μικρό αριθμό νέων στόχων. Αν από το πρόγραμμα φαίνεται, πως η υλοποίηση ενός επιπέδου θα απαιτούσε ένα εξαιρετικά μεγάλο αριθμό νέων υποστόχων, τότε είναι καλύτερα να εκβαθύνουμε την υλοποίηση. Δηλαδή να δημιουργήσουμε ένα παραπάνω επίπεδο υποστόχων περιορίζοντας ταυτόχρονα των αριθμό των υποστόχων ανά επίπεδο.

Σημαντικό πλεονέκτημα του σχεδιασμού top-down είναι οι αποφάσεις που αναβάλλουμε για αργότερα. Ακολουθώντας τον σχεδιασμό αυτό, δεν είναι αναγκαίο να πάρουμε αποφάσεις μέχρι να κωδικοποιήσουμε τα κομμάτια τα οποία αφορούν.

Οι δομές δεδομένων αναπτύσσονται επίσης με μία top-down μέθοδο. Οι αρχικές αποφάσεις κατευθύνονται από τις ανάγκες που έχουμε για συγκεκριμένες δομές δεδομένων με υψηλό τρόπο οργάνωσης. Οι αποφάσεις χαμηλών επιπέδων μπορούν να αναβληθούν για μετέπειτα εκλεπτύνσεις.

Ένα άλλο πλεονέκτημα της μεθόδου σχεδιασμού top-down είναι πως καθιστούμε έγκυρη κάθε μονάδα προγράμματος (program unit) καθώς αυτό αναπτύσσεται. Δεν είναι ο αριθμός των εντολών ο μόνος παράγοντας που επιδρά στον χρόνο που απαιτείται για την αποσφαλμάτωση. Τα αποτελέσματα των επιδράσεων μεταξύ των εντολών έχει αρκετά μεγάλη συνεισφορά στο πρόβλημα. Αυτό σημαίνει πως ο χρόνος t που απαιτείται για να αποσφαλματωθεί ένα πρόγραμμα που αποτελείται από n γραμμές δεν αυξάνεται με τον ίδιο ρυθμό που αυξάνεται το μήκος του προγράμματος αλλά ταχύτερα. Η σχέση μεταξύ t και n δίνεται προσεγγιστικά από τον τύπο:

$t \approx n^k. \quad k > 1$

Όταν το πρόγραμμα γίνει πολύ μεγάλο τότε ο χρόνος αποσφαλμάτωσης αυξάνει δραματικά με αποτέλεσμα να καταναλώνει το 50 με 70% του συνολικού χρόνου που απαιτείται για το πρόγραμμα (project).

Η αντίθετη στρατηγική καλείται από την ειδική προς την γενική (bottom-up) μέθοδος. Σε μια bottom-up μέθοδο, γράφουμε πρώτα τους ορισμούς των συναρτήσεων πριν τη χρήση των αντίστοιχων κλήσεων τους μέσα σε υψηλότερου επιπέδου συναρτήσεις. Αυτό μας αναγκάζει να δώσουμε τις λεπτομέρειες των κατώτατων στρωμάτων του συνολικού προγράμματος πριν τη συμπλήρωση τους στα υψηλότερα επίπεδα που χρησιμοποιούν τις συναρτήσεις που έχουμε ήδη γράψει στα χαμηλά επίπεδα. Έτσι, στον bottom-up προγραμματισμό, ορίζουμε συναρτήσεις πριν τις χρησιμοποιήσουμε, ενώ στον top-down προγραμματισμό, τις χρησιμοποιούμε προτού τις ορίσουμε.

7. ΔΙΑΣΠΑΣΗ ΠΡΟΓΡΑΜΜΑΤΟΣ ΣΕ ΔΟΜΙΚΑ ΣΤΟΙΧΕΙΑ (PROGRAM MODULARITY)

Ένα καλοσχεδιασμένο δομικό στοιχείο (module) σε ένα πρόγραμμα πρέπει να έχει τα εξής χαρακτηριστικά:

1. Διαχωρισμός μεταξύ διεπαφών και υλοποίησης.
2. Ενθυλάκωση (Encapsulation) δεδομένων.
3. Συστηματική διαχείριση πόρων του προγράμματος.
4. Συνεπής χρήση ονομάτων.
5. Απλή διεπαφή διαχείρισης.
6. Επαρκής αναφορά των λαθών.
7. Περιέχει συμβάσεις.
8. Μεγάλη συνοχή στα επιμέρους στοιχεία του.
9. Ασθενή/μικρή εξάρτηση με τα υπόλοιπα modules στο πρόγραμμα.

7.1 Διαχωρισμός μεταξύ διεπαφών και υλοποίησης.

Η χρήση των διεπαφών κρύβει την υλοποίηση από τους πελάτες/χρηστές ενισχύοντας έτσι την αφαιρετικότητα του προγράμματος. Επίσης δίνει την δυνατότητα για ξεχωριστή μεταγλώττιση των διαφορετικών υλοποιήσεων του πηγαίου κώδικα.

Έστω για παράδειγμα ότι προσπαθούμε να υλοποιήσουμε μια στοίβα που τα στοιχεία της είναι συμβολοσειρές (strings). Για την υλοποίηση χρησιμοποιούμε μια συνδεδεμένη λίστα με τις παρακάτω λειτουργίες:

- new : δημιουργεί και επιστρέφει ένα νέο αντικείμενο στοίβας
- free : αποδεσμεύει ένα δεδομένο αντικείμενο στοίβας
- push : ωθεί μια δεδομένη συμβολοσειρά στην κορυφή της στοίβας
- pop : εξάγει μια συμβολοσειρά από την κορυφή της στοίβας
- top : επιστρέφει το στοιχείο στην κορυφή της στοίβας
- isEmpty : επιστρέφει 1(αληθές) αν η στοίβα είναι κενή αλλιώς 0 (ψευδές)

Παράδειγμα 1

```
/* stack.c */
struct Node {
      const char *item;
      struct Node *next;
};
struct Stack {
      struct Node *first;
};
struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
void  Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}

/* client.c */
#include "stack.c"
/* Use the functions
defined in stack.c. */
```

Η υλοποίηση της στοίβας αποτελείται από ένα αρχείο stack.c, χωρίς διεπαφή.

Ο τρόπος που το πρόγραμμα πελάτης συνδέεται με το stack.c (#include "stack.c") μπορεί να προκαλέσει τα εξής θέματα:

- Σε περίπτωση αλλαγής κώδικα στο stack.c πρέπει να ξαναχτίσουμε το stack.c μαζί με το πρόγραμμα του πελάτη.
- Ο πελάτης έχει πρόσβαση στην υλοποίηση της στοίβας (κακή αφαιρετικότητα).

Παράδειγμα 2

```
/* stack.h */
struct Node {
     const char *item;
     struct Node *next;
};
struct Stack {
     struct Node *first;
};
struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);

/* client.c */
#include "stack.h"
/* Use the functions declared in stack.h. */
```

Η υλοποίηση της στοίβας αποτελείται από δύο αρχεία :

1. Το stack.h (η διεπαφή): που δηλώνει συναρτήσεις και ορίζει δομές δεδομένων που θα χρησιμοποιηθούν.
2. Το stack.c (η υλοποίηση): που ορίζει συναρτήσεις, περιέχει και το stack.h (#includes) ώστε ο μεταγλωττιστής να μπορεί να ελέγξει για συνέπεια μεταξύ δηλώσεων και ορισμών και οι συναρτήσεις να έχουν πρόσβαση στις απαραίτητες δομές.

Ο πελάτης τώρα συνδέεται μόνο με τη διεπαφή (#include "stack.h") επιτυγχάνοντας τα παρακάτω:

- Όταν αλλάζουμε τον κώδικα στο stack.c πρέπει να ξαναχτίσουμε **μόνο** το αρχείο αυτό και **όχι** το πρόγραμμα του πελάτη.
- Ο πελάτης δεν βλέπει τις υλοποιήσεις των συναρτήσεων (καλύτερη αφαιρετικότητα και απόκρυψη).

Ένα σωστό παράδειγμα διεπαφής με σωστή απόκρυψη και αφαιρετικότητα είναι και η διεπαφή για τις συμβολοσειρές (string) :

```
/* string.h */
size_t strlen(const char *s);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strcat(char *dest, const char *src);
char  *strncat(char *dest, const char *src, size_t n);
char  *strcmp(const char *s, const char *t);
char  *strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
…
```

7.2 Ενθυλάκωση (Encapsulation) δεδομένων.

Ένα καλοσχεδιασμένο δομικό στοιχείο ενθυλακώνει σωστά τα δεδομένα χρησιμοποιώντας τις εσωτερικές του συναρτήσεις. Η διεπαφή του θα πρέπει να κρύβει όλες τις λεπτομέρειες της υλοποίησης. Επίσης δεν θα πρέπει να επιτρέπεται η απ' ευθείας πρόσβαση και επεξεργασία των δεδομένων από τα προγράμματα των πελατών. Με αυτόν τον τρόπο επιτυγχάνουμε μεγαλύτερη σαφήνεια στον κώδικα μας και ενθαρρύνουμε την αφαιρετικότητα. Επίσης εξασφαλίζουμε μεγαλύτερη ασφάλεια καθώς τα προγράμματα πελάτες δεν μπορούν να αλλοιώσουν τα αντικείμενα του δομικού μας στοιχείου και να αλλάξουν τα δεδομένα τους με απρόβλεπτους τρόπους. Τέλος επιτυγχάνουμε και καλύτερη ελαστικότητα στην υλοποίηση μας. Αυτό μας επιτρέπει να κάνουμε αλλαγές στην υλοποίηση μας χωρίς να επηρεάζονται τα προγράμματα – πελάτες.

Έστω το παρακάτω παράδειγμα υλοποίησης στοίβας :

```
/* stack.h */
struct Node {
    const char *item;
    struct Node *next;
};

struct Stack {
    struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Εδώ έχουμε τον ορισμό του τύπου δεδομένων που χρησιμοποιούμε μέσα στο αρχείο stack.h. Έτσι επιτυγχάνουμε μερική απόκρυψη. Η διεπαφή όμως φανερώνει πως έγινε η υλοποίηση (δηλαδή με τη χρήση συνδεδεμένης λίστας). Τέλος ο πελάτης έχει πρόσβαση και μπορεί να αλλάξει τα δεδομένα απευθείας με κίνδυνο να αλλοιώσει τα αντικείμενα.

Στο παρακάτω παράδειγμα έχουμε κάνει κάποιες αλλαγές:

```
/* stack.h */
struct Stack;
struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Εδώ μεταφέραμε την υλοποίηση του τύπου δεδομένων μας στο αρχείο stack.c και αφήσαμε μόνο μία δήλωση του ονόματος του τύπου στοίβας. Με αυτή την αλλαγή κρύψαμε τον τρόπο υλοποίησης της στοίβας. Επίσης το πρόγραμμα - πελάτης δεν έχει απ' ευθείας πρόσβαση στα δεδομένα πλέον.

Ακολουθεί ένα νέο παράδειγμα μετά από κάποιες επιπλέον αλλαγές :

```
/* stack.h */
typedef struct Stack * Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

Εδώ χρησιμοποιούμε ένα αδιαφανή δείκτη για τη δομή μας (Stack_T). Η διεπαφή τώρα παρέχει στον πελάτη μια συντόμευση για την δομή. Επίσης η διεπαφή ενθαρρύνει το πρόγραμμα - πελάτη να μεταχειρίζεται την στοίβα σαν ένα αντικείμενο και όχι σαν δείκτη σε δομή. Το πρόγραμμα πελάτης εξακολουθεί να μην έχει πρόσβαση στα δεδομένα.

Ένα άλλο παράδειγμα ενθυλάκωσης που συναντούμε συχνά είναι και η εξής δομή για την διαχείριση των αρχείων:

```
/* stdio.h */
struct FILE {
    int cnt;    /* characters left */
    char *ptr;  /* next character position */
    char *base; /* location of buffer */
    int flag;   /* mode of file access */
    int fd;     /* file descriptor */
};
```

Η συγκεκριμένη υλοποίηση για της δομή του αρχείου αντιτίθεται στα θετικά στοιχεία που αναφέραμε παραπάνω. Δηλαδή, οι προγραμματιστές έχουν πρόσβαση στα δεδομένα απ' ευθείας και μπορούν να αλλοιώσουν τα αντικείμενα ή και να γράψουν κώδικα που δεν θα είναι συμβατός με όλα τα συστήματα. Οι συναρτήσεις όμως έχουν οριστεί καλά με επαρκής τεκμηρίωση. Με αυτό τον τρόπο πολύ λίγοι προγραμματιστές που ασχολούνται εκτενώς με το stdio.h θα μπουν σε πειρασμό να αποκτήσουν πρόσβαση απ' ευθείας στα δεδομένα.

7.3 Συστηματική διαχείριση πόρων του προγράμματος.

Ένα καλοσχεδιασμένο δομικό στοιχείο διαχειρίζεται τους πόρους του με συνέπεια. Η αποδέσμευση των πόρων θα πρέπει να γίνεται μόνο από το στοιχειό που τους είχε δεσμεύσει αρχικά. Ένα αντικείμενο για παράδειγμα όταν δεσμεύει προσωρινή μνήμη θα πρέπει το ίδιο να την αποδεσμεύει. Επίσης όταν ανοίγει ένα αρχείο για να το χρησιμοποιήσει θα πρέπει το ίδιο να το κλείνει μετά.

Η διαδικασία της δέσμευσης και αποδέσμευσης πόρων σε διαφορετικά επίπεδα του προγράμματος είναι επιρρεπής σε λάθη όπως:

- Διαρροή μνήμης (Memory leaks) που προκύπτουν όταν ξεχνάμε να αποδεσμεύσουμε τη μνήμη που δε χρειαζόμαστε.
- Σφάλματα κατάμτησης (Segmentation faults) από μετέωρους/λανθασμένους δείκτες, όταν ξεχνάμε να δεσμεύσουμε μνήμη που χρειαζόμαστε.
- Αναποτελεσματική χρήση περιορισμένων πόρων όπως περιγραφέων αρχείων (file descriptors) του συστήματος που προκύπτουν όταν ξεχνάμε να κλείνουμε τα αρχεία που χρησιμοποιήσαμε.
- Segmentation faults από λάθος τιμές σε δείκτες αρχείων, που προκύπτουν όταν ξεχνάμε να ανοίξουμε ένα αρχείο που χρειαζόμαστε.

Χρησιμοποιώντας για παράδειγμα τη στοίβα μας που περιγράψαμε παραπάνω προκύπτουν οι εξής λογικές επιλογές:

1. Το πρόγραμμα πελάτης δεσμεύει και αποδεσμεύει χώρο για τις συμβολοσειρές.

Με αυτό τον τρόπο επιτυγχάνουμε τα εξής:

- Η **Stack_push()** δεν δημιουργεί αντίγραφα για τις δεδομένες συμβολοσειρές.
- Η **Stack_pop()** δεν αποδεσμεύει την εξαγόμενη συμβολοσειρά.
- Η **Stack_free()** δεν αποδεσμεύει τα υπόλοιπα στοιχεία της στοίβας.

2. Το ίδιο το αντικείμενο της στοίβας δεσμεύει και αποδεσμεύει το χώρο. Αυτό θα έχει ως συνέπεια τα εξής :

- Η **Stack_push()** δημιουργεί αντίγραφα για τις δεδομένες συμβολοσειρές.
- Η **Stack_pop()** αποδεσμεύει την εξαγόμενη συμβολοσειρά.
- Η **Stack_free()** αποδεσμεύει όλα τα υπόλοιπα στοιχεία της στοίβας.

Η επιλογή μας προς υλοποίηση εδώ θα ήταν η 1 αλλά αυτό είναι κάτι συζητήσιμο. Κάποιες διαφορετική επιλογές από τις παραπάνω θα ήταν και το πρόγραμμα - πελάτης να δεσμεύει τη μνήμη και το αντικείμενο της στοίβας την αποδεσμεύει ή το αντίστροφο. Είναι προφανές ότι κάτι τέτοιο θα ήτανε λάθος.

Αν επανέλθουμε πάλι στο παράδειγμα για το string.h παρατηρούμε τα εξής :

- Το δομικό στοιχείο είναι Stateless.
- Δεν έχει καθόλου πόρους για διαχείριση.

Ενώ για το stdio.h βλέπουμε μια διαφορετική προσέγγιση. Εδώ η **fopen()** δεσμεύει μνήμη και χρησιμοποιεί περιγραφείς αρχείων ενώ ή **fclose()** αποδεσμεύει μνήμη και ελευθερώνει τους περιγραφείς αρχείων

Γενικά σε περιπτώσεις που χρειάζεται να παραβούμε τις παραπάνω οδηγίες και να μεταφέρουμε την κατοχή των πόρων στον πελάτη, θα πρέπει να το αναφέρουμε ρητά στα σχόλια της συνάρτησης.

Π.χ.

somefile.h

```
void *f(void);
/*   This function allocates memory for the returned object.
      You (the caller) own that memory, and so are responsible
       for freeing it when you no longer need it.
*/
```

7.4 Συνεπής χρήση ονομάτων.

Ένα καλοσχεδιασμένο δομικό στοιχείο θα πρέπει να είναι και συνεπής. Τα ονόματα των συναρτήσεων που έχει θα πρέπει να αναφέρονται στο ίδιο το στοιχείο. Αυτό εξυπηρετεί στην διαδικασία συντήρησης του προγράμματος καθώς οι προγραμματιστές μπορούν πιο εύκολα να εντοπίσουν τις συναρτήσεις. Επίσης εξαλείφει την πιθανότητα να προκύψουν συγκρούσεις μεταξύ ονομάτων (από διαφορετικούς προγραμματιστές κ.τ.λ.π.)

Οι συναρτήσεις ενός δομικού στοιχείου θα πρέπει να χρησιμοποιούν μια σταθερή, συνεπή σειρά παραμέτρων για να διευκολύνουν τη συγγραφή του κώδικα του πελάτη. Ακολουθούν μερικά θετικά παραδείγματα :

Για τη στοίβα μας (Stack) :

- κάθε όνομα συνάρτησης ξεκινά με το πρόθεμα “Stack_”.

- η πρώτη παράμετρος της κάθε συνάρτησης αντιστοιχεί στο αντικείμενο της στοίβας.

Για το δομικό στοιχείο string :

- κάθε όνομα συνάρτησης ξεκινά με το πρόθεμα “str”.
- η παράμετρος για τη συμβολοσειρά προορισμού (destination) είναι πάντοτε πριν τη συμβολοσειρά πηγής (source).

Και ένα αρνητικό παράδειγμα για το δομικό στοιχείο stdio :

- μερικές συναρτήσεις ξεκινούν με το πρόθεμα “f”, ενώ άλλες όχι.
- μερικές συναρτήσεις χρησιμοποιούν την πρώτη παράμετρο το αντικείμενο τύπου αρχείου (FILE) ενώ άλλες, όπως η putc() χρησιμοποιούν διαφορετικές παραμέτρους.

7.5 Απλή διεπαφή διαχείρισης

Ένα καλοσχεδιασμένο δομικό στοιχείο έχει την ποιο απλή και μινιμαλιστική διεπαφή. Η δήλωση συναρτήσεων θα πρέπει να γίνεται στη διεπαφή ενός δομικού στοιχείου μόνο αν η συνάρτηση είναι απαραίτητη για να συμπληρώσει το αντικείμενο και μόνο αν είναι εύχρηστη για πολλούς χρήστες. Και αυτό γιατί διατηρώντας περισσότερες συναρτήσεις αυξάνεται το κόστος εκμάθησης και συντήρησης του ίδιου του δομικού στοιχείου.

Αν παρατηρήσουμε τη στοίβα μας, για παράδειγμα, θα διαπιστώσουμε ότι όλες οι συναρτήσεις είναι απαραίτητες.

```
/* stack.h */
typedef struct Stack *Stack_T ;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

Θα μπορούσαμε να προσθέσουμε μια επιπλέον συνάρτηση, την:

```
void Stack_clear(Stack_T s);
```

Αυτή θα είχε ως σκοπό να εξάγει όλα τα στοιχεία από ένα αντικείμενο στοίβας (να την αδειάσει δηλαδή). Μια τέτοια συνάρτηση όμως δεν είναι εντελώς απαραίτητη γιατί ένα πρόγραμμα – πελάτη αφού η ίδια λειτουργικότητα μπορεί να υλοποιηθεί εξάγοντας ένα - ένα τα στοιχεία επανειλημμένα με την χρήση την pop. Για κάποιο άλλο πρόγραμμα όμως θα μπορούσε να είναι χρήσιμη.

Για το προηγούμενο παράδειγμα του στοιχείου string αυτές οι εντελώς απαραίτητες συναρτήσεις είναι οι:

```
/* string.h */

size_t strlen(const char *s);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strncat(char *dest, const char *src, size_t n);
char  *strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
```

και οι απλώς βολικές και εύχρηστες συναρτήσεις είναι οι:

```
char  *strcpy(char *dest, const char *src);
char  *strcat(char *dest, const char *src);
char  *strcmp(const char *s, const char *t);
```

Παρατηρούμε ότι οι συναρτήσεις που δεν έχουν την παράμετρο **n** φαίνεται να είναι πιο εύχρηστες.

Για το παράδειγμα του στοιχείου stdio έχουμε οι εντελώς απαραίτητες συναρτήσεις είναι οι:

```
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);
int   fgetc(FILE *f);
int   getc(FILE *f);
int   putc(int c, FILE *f);
int   fscanf(FILE *f, const char *format, );
int   fprintf(FILE *f, const char *format, );
```

και οι βολικές και εύχρηστες συναρτήσεις είναι οι:

```
int   getchar(void);
int   putchar(int c);
int   scanf(const char *format, );
int   printf(const char *format, );
```

Το κατά πόσο εύχρηστες είναι οι παραπάνω συναρτήσεις όμως είναι θέμα επιλογής του κάθε προγραμματιστή.

7.6 Επαρκής αναφορά των λαθών.

Ένα καλοσχεδιασμένο δομικό στοιχείο πρέπει να αναφέρει τα λάθη στα προγράμματα - πελάτες χωρίς πολλές υπερβολές. Θα πρέπει δηλαδή να ανιχνεύει τα λάθη αλλά και να επιτρέπει στα προγράμματα - πελάτες να τα διαχειριστούν Η διαχείριση λαθών επιτρέπει περισσότερη ευελιξία στο πρόγραμμα. Φυσικά μέσα στο δομικό στοιχείο δεν θα πρέπει να κάνουμε παραδοχές για το τι είδος βλαβοληπτικής ενέργειας θα προτιμήσει ο πελάτης.

Στη γλώσσα C έχουμε συνήθως δυο επιλογές για να ανιχνεύουμε λάθη:

- Τη χρήση της εντολής if <statement>
- Τη χρήση της μακρο-εντολής assert

Για να αναφέρουμε τα λάθη στα προγράμματα - πελάτες όμως έχουμε περισσότερες επιλογές όπως :

- Ανάθεση τιμής σε καθολική μεταβλητή. Η προσέγγισή αυτή συνήθως προκαλεί προβλήματα σε πολυνηματικές εφαρμογές. Επίσης είναι πολύ συνηθισμένο τα προγράμματα – πελάτες να ξεχνούν να την ελέγξουν.
- Χρήση της επιστρεφόμενης τιμής μιας συνάρτηση. Η επιλογή αυτή δεν εξυπηρετεί πάντοτε, ειδικά όταν η τιμή που επιστρέφεται έχει άλλη φυσική σημασία για τον πελάτη. Έστω για παράδειγμα αν σε μια συνάρτηση που επιστέφει ένα δεκαδικό αριθμό αυτό χρησιμοποιείτε σαν χρηματικό ποσό. Αν επιλέξουμε να επιστρέψουμε 0 σε περίπτωση λάθους αυτό μπορεί να μην εξυπηρετεί το πρόγραμμα που θα χρησιμοποιήσει αυτή τη συνάρτηση καθώς η τιμή 0 θα μπορούσε να ερμηνευτεί και σαν 0 ποσό και όχι σας κάποιο λάθος.
- Χρήση επιπλέον παραμέτρου στην κλήση της συνάρτησης. Με αυτό τον τρόπο όμως γίνεται πιο δύσχρηστη η συνάρτηση. Το πρόγραμμα - πελάτης θα πρέπει να προβλέψει για επιπλέον παραμέτρους και πιθανώς να χρειάζεστε και επιπλέον πόρους.

- Χρήση μακροεντολής assert. Το κυριότερο μειονέκτημά αυτής της μεθόδου είναι ότι σε περίπτωση λάθους τερματίζει και το πρόγραμμα του πελάτη.

Είναι προφανές ότι καμία από τις παραπάνω επιλογές δεν είναι η ιδανική. Θα μπορούσαμε να κάνουμε όμως έναν επιπλέον διαχωρισμό σε λάθη που προκαλούνται από το χρήστη και σε λάθη που προκαλούνται από τον προγραμματιστή και να τα αντιμετωπίσουμε αναλόγως.

Συνηθισμένα χαρακτηριστικά για τα λάθη του χρήστη είναι και τα εξής :

- Λάθη που προκαλούνται από ανθρώπινους χρήστες όπως λάθος δεδομένα για είσοδο ή χρήση λάθος παραμέτρων σε μια εντολή.
- Λάθη που είναι εύκολο να συμβούν.
- Για να τα εντοπίσουμε συνήθως χρησιμοποιούμε ελέγχους με if.
- Για να τα αναφέρουμε χρησιμοποιούμε τιμές που επιστρέφουν οι συναρτήσεις ή παραμέτρους που περνάμε σαν δείκτες μέσα στη συνάρτηση

Ενώ τα συνηθισμένα χαρακτηριστικά για τα λάθη προγραμματιστή είναι :

- Λάθη που γίνονται από τους ίδιους τους προγραμματιστές συνήθως κατά την κατασκευή του προγράμματος.
- Λάθη που δεν θα έπρεπε να συμβαίνουν κανονικά κάτω από οποιεσδήποτε συνθήκες, όπως π.χ. η κλήση μιας stack_pop() όταν η στοίβα είναι κενή ή δεν έχει οριστεί.
- Για να τα ανιχνεύουμε και να τα αναφέρουμε χρησιμοποιούμε συνήθως την assert.

Αυτός ο διαχωρισμός μεταξύ των κατηγοριών λαθών δεν είναι εύκολο να γίνει πάντοτε. Σε μερικές περιπτώσεις για παράδειγμα όταν μια εγγραφή σε ένα αρχείο στο δίσκο αποτυγχάνει γιατί ο δίσκος είναι γεμάτος αυτό μπορεί να θεωρηθεί και σαν λάθος του χρήστη που δεν είχε τους απαραίτητους πόρους να τρέξει το πρόγραμμα, αλλά και σαν λάθος προγραμματιστή που δεν διαχειρίστηκε σωστά τη διαδικασία εγγραφής στο δίσκο.

Στη στοίβα μας για παράδειγμα διακρίνουμε τις παρακάτω περιπτώσεις λάθους και τις αντιμετωπίζουμε αναλόγως.

```
/* stack.c */

void Stack_push(Stack_T s, const char *item) {
    struct Node *p;
    assert(s != NULL);
    p = (struct Node*)malloc(sizeof(struct Node));
    assert(p != NULL);
    p->item = item;
    p->next = s->first;
    s->first = p;
}
```

Θεωρούμε ότι η κλήση της συνάρτησης **Stack_push**() με λάθος παράμετρο **s** είναι λάθος του προγραμματιστή για αυτό και χρησιμοποιούμε την assert για τον έλεγχο. Το ίδιο γίνεται επίσης και για την τιμή της μεταβλητής **p** μετά την κλήση της **malloc()** όπου η αποτυχία της **malloc()** θεωρείται λάθος του προγραμματιστή. Τέλος δεν ελέγχουμε ούτε αναφέρουμε κανένα λάθος του χρήστη.

Για το δομικό στοιχείο string δεν γίνεται κανένας έλεγχος ή αναφορά σφαλμάτων. Αν για παράδειγμα περάσουμε σαν παράμετρο μια κενή (null) συμβολοσειρά θα προκύψει ένα segmentation fault. Αντίθετα για το δομικό στοιχείο stdlib, γίνεται επιστροφή τιμών που

υποδηλώνουν τα λάθη. Επίσης χρησιμοποιείται και η καθολική μεταβλητή errno με τον αντίστοιχο κωδικό του σφάλματος.

7.7 Περιέχει συμβάσεις.

Ένα καλοσχεδιασμένο δομικό στοιχειό πρέπει να ορίζει συμβάσεις με τους πελάτες του. Οι συμβάσεις αυτές θα πρέπει να περιγράφουν αναλυτικά τη λειτουργιά κάθε συνάρτησης. Δηλαδή χαρακτηριστικά όπως τα παρακάτω:

- Τι σημαίνουν οι παράμετροι της.
- Ποιες είναι οι επιτρεπτές τιμές που θα πάρουν και ποιες είναι οι μη επιτρεπτές.
- Τι τιμές θα επιστρέφουν και
- τυχόν παρενέργειες που θα προκύψουν.

Οι συμβάσεις αυτές διευκολύνουν τη συνεργασία μεταξύ πολλών προγραμματιστών στην ίδια ομάδα. Επίσης βοηθούν στον καλύτερο εντοπισμό των λαθών και στην ανάθεση ευθυνών για τα λάθη στα σωστά άτομα.

Για να το πετύχουμε αυτό στην στοίβα μας χρησιμοποιούμε τα παρακάτω σχόλια στον κώδικά μας:

```
/* stack.h */

char *Stack_top(Stack_T s);
/* Return the top item of stack s.
It is a checked runtime error for s
to be NULL or empty. */
 …
```

Με τα σχόλια μπορούμε να δούμε το νόημα κάθε παραμέτρου. Η παράμετρος **s** δηλαδή είναι το σχετικό αντικείμενο της στοίβας μας. Για να καθορίσουμε τις έγκυρες και μη τιμές πάλι με σχόλια αναφέρουμε ότι το s δεν μπορεί να είναι κενό ή null. Επίσης η σημασία της τιμής που επιστρέφεται είναι ότι πρόκειται για το κορυφαίο στοιχείο της στοίβας. Τέλος αναφέρουμε ότι δεν υπάρχουν καθόλου παρενέργειες.

7.8 Μεγάλη συνοχή στα επιμέρους στοιχεία του.

Ένα καλοσχεδιασμένο δομικό στοιχείο έχει ισχυρή συνάφεια καθώς οι συναρτήσεις του συνδέονται ισχυρά αναμεταξύ τους. Αυτό βοήθα στην καλύτερη αφαιρετικότητα του δομικού στοιχείου.

Στη στοίβα μας για παράδειγμα επιτυγχάνουμε συνάφεια γιατί όλες οι συναρτήσεις που συσχετίζονται με τα εμφωλευμένα δεδομένα μας. Στο δομικό στοιχείο string που αναφερθήκαμε και νωρίτερα παρατηρούμε τα εξής:

- Οι περισσότερες συναρτήσεις αναφέρονται στη διαχείριση συμβολοσειρών (θετικό).
- Μερικές συναρτήσεις (**memcpy()**, **memmove()**, **memcmp()**, **memchr()**, **memset())** δεν συσχετίζονται με τη διαχείριση συμβολοσειρών (αρνητικό).
- Οι συναρτήσεις αυτές όμως είναι παρόμοιες με τις συναρτήσεις διαχείρισης συμβολοσειρών (θετικό).

Παράλληλα στο δομικό στοιχείο stdio έχουμε τα εξής χαρακτηριστικά:

- Οι περισσότερες συναρτήσεις συσχετίζονται με τα ρεύματα εισόδου/εξόδου (I/O) (θετικό).
- Μερικές συναρτήσεις όμως όχι, όπως οι **sprintf()**, **sscanf()** (αρνητικό).

- Οι συναρτήσεις αυτές όμως είναι παρόμοιες με τις παραπάνω (θετικό).

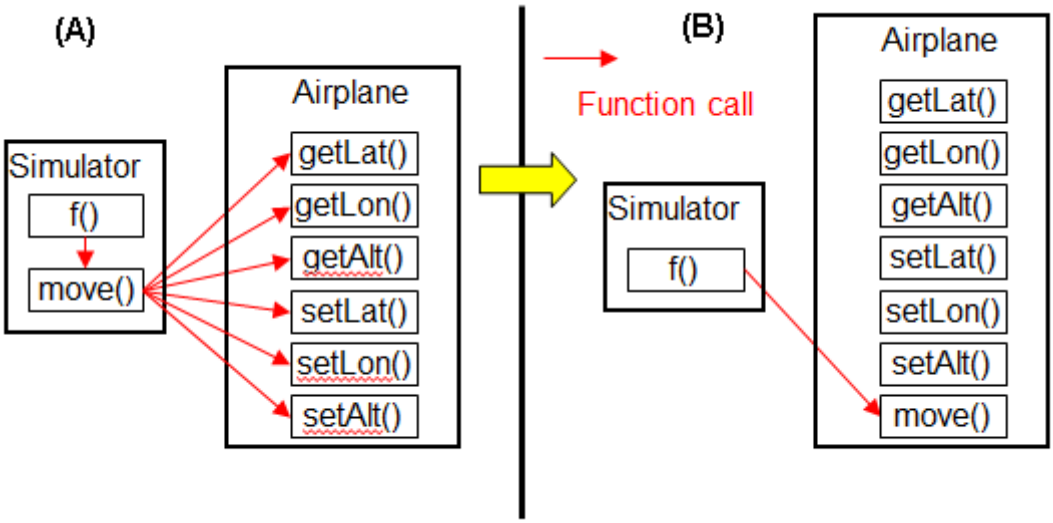
7.9 Ασθενής/μικρή εξάρτηση με τα υπόλοιπα modules στο πρόγραμμα.

Ένα καλοσχεδιασμένο δομικό στοιχείο έχει ασθενή σύνδεση με τα υπόλοιπα δομικά στοιχεία του προγράμματος. Οι αλληλεπιδράσεις ανάμεσα στα μέρη του δομικού στοιχείου θα πρέπει να είναι πιο ισχυρές από τις αλληλεπιδράσεις με τα άλλα δομικά στοιχεία. Αυτό βοηθά στη καλύτερη συντήρηση και αντικατάσταση των τμημάτων του προγράμματος όπως και στην επαναχρησιμοποίηση των δομικών του στοιχείων. Πρακτικά φαίνεται ότι τα δομικά στοιχεία που είναι ασθενέστερα συνδεδεμένα παρουσιάζουν λιγότερα σφάλματα.

Για να πετύχουμε ασθενή σύνδεση μπορούμε να δοκιμάσουμε κάποιες από τις παρακάτω τεχνικές. Μπορούμε για παράδειγμα να μεταφέρουμε κώδικα από το πρόγραμμα – πελάτη στο ίδιο το δομικό στοιχείο. Μπορούμε αντίστροφα να μεταφέρουμε και κώδικα από το δομικό στοιχείο στο πρόγραμμα πελάτη. Τέλος μπορούμε να μεταφέρουμε κώδικα από το δομικό στοιχείο και το πρόγραμμα πελάτη σε ένα νέο δομικό στοιχειό.

Στα σχήματα που ακολουθούν εξηγούμε καλύτερα τους διάφορους τρόπους σύνδεσης.

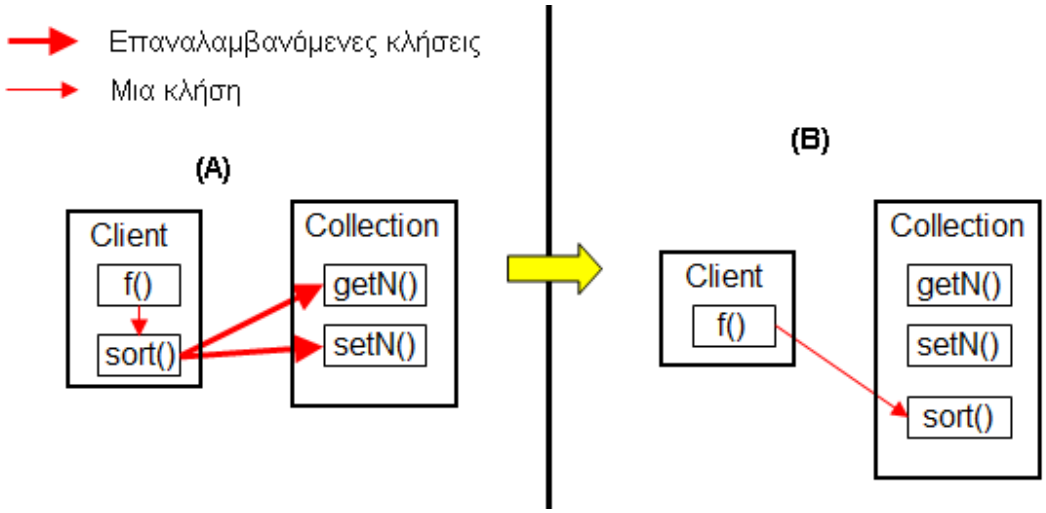
7.9.1 Design – time coupling



Σχήμα 10: Design time coupling (Ζύζευξη).

1. Το πρόγραμμα – πελάτη (simulator) καλεί πολλές διαφορετικές συναρτήσεις από το δομικό μας στοιχείο (Airplane), οπότε προκύπτει ισχυρή σύνδεση.
2. Έχοντας μεταφέρει τη συνάρτηση move() στο δομικό μας στοιχείο, το πρόγραμμα – πελάτη πλέον κάνει λίγες κλήσεις. Με αυτό τον τρόπο επιτυγχάνουμε ασθενή σύνδεση.

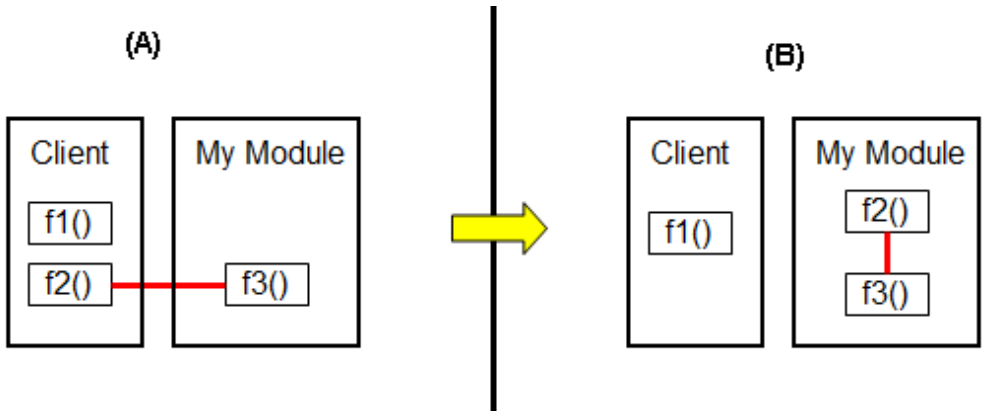
7.9.2 Run – time coupling



Σχήμα 11: Run time coupling.

1. Το πρόγραμμα – πελάτης (Client) καλεί πολλαπλές κλήσεις στις ίδιες συναρτήσεις του δομικού μας στοιχείο (Collection), οπότε προκύπτει ισχυρή σύνδεση.
2. Έχοντας μεταφέρει τη συνάρτηση sort() στο δομικό μας στοιχείο, το πρόγραμμα – πελάτης πλέον κάνει μια μόνο κλήση. Με αυτό τον τρόπο επιτυγχάνουμε ασθενή σύνδεση.

7.9.3 Maintenance – time coupling



Σχήμα 12: Maintenance time coupling.

1. Ο προγραμματιστής χρειάζεται να κάνει συχνά αλλαγές και στο πρόγραμμα – πελάτη (Client) και στο δομικό στοιχείο (My Module). Οπότε προκύπτει ισχυρή σύνδεση.
2. Ο προγραμματιστής χρειάζεται να κάνει αλλαγές στο δομικό στοιχείο συχνά αλλά όχι τόσο συχνά στο πρόγραμμα πελάτη. Με αυτό τον τρόπο επιτυγχάνουμε ασθενή σύνδεση.

8. GENERICS (ΓΕΝΙΚΟΤΗΤΑ)

Στο κεφάλαιο αυτό θα επεκτείνουμε την έννοια των δομικών στοιχείων που αναφέραμε πριν και θα μιλήσουμε για τα generic modules (γενικού τύπου δομικά στοιχεία). Ο σκοπός εδώ είναι να μπορούμε να κατασκευάζουμε δομικά στοιχεία που να χρησιμοποιούνται με πολλούς διαφορετικούς τύπους δεδομένων. Επίσης θα αναπτύξουμε συναρτήσεις που θα μπορούν να δουλεύουν με όλους τους διαφορετικούς τύπους δεδομένων.

Το όφελος εδώ είναι σημαντικό. Χρησιμοποιώντας generic modules μπορούμε να επαναχρησιμοποιούμε τμήματα του υπάρχοντος μας πηγαίου κώδικά. Με αυτό τον τρόπο κερδίζουμε χρόνο από την κατασκευή ενός νέου τμήματος κώδικα και μειώνουμε το συνολικό κόστος κατασκευής του προγράμματος. Γενικά τα generic modules είναι πιο εύχρηστα και μπορούν να χρησιμοποιηθούν ξανά σε σχέση με τα μη generic.

8.1 Παράδειγμα Generic Data Structure

Έστω η παρακάτω υλοποίηση για τον ΑΤΔ στοίβα

```
/* stack.h */
typedef struct Stack *Stack_T;
Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int  Stack_isEmpty(Stack_T s);
```

Είναι προφανές ότι τα στοιχεία που επεξεργάζεται η παραπάνω δομή είναι συμβολοσειρές (strings). Οι γενικές λειτουργίες του ΑΤΔ τύπου στοίβας όμως μπορούν χρησιμοποιηθούν και για διαφορετικούς τύπους δεδομένων πέρα από τις συμβολοσειρές. Από αυτό καταλαβαίνουμε ότι θα είχε περισσότερο νόημα η υλοποίηση μας για τον ΑΤΔ στοίβα να είναι πιο γενική. Το πρόβλημα που αντιμετωπίζουμε λοιπόν είναι πώς να μετατρέψουμε αυτή την υλοποίηση ώστε να μπορούμε να χρησιμοποιούμε αυτή την δομή για να αποθηκεύουμε δεδομένα οποιουδήποτε τύπου.

Μια πιθανή λύση είναι να επιτρέπουμε στα προγράμματα πελάτης να ορίζουν αυτά τον τύπο δεδομένων που θα χρησιμοποιεί η στοίβα. Μπορούμε να δηλώσουμε δηλαδή ένα τύπο δεδομένων **Item_T** στο header αρχείο μας (**stack.h**) και στο πρόγραμμά πελάτη (**client.c**) να προσθέσουμε την υλοποίηση του:

```
/* stack.h */
typedef struct Item *Item_T;
typedef struct Stack *Stack_T;
Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
void    Stack_push(Stack_T s, Item_T item);
Item_T  Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

```
/* client.c */
struct Item {
     char *str; /* Or whatever is appropriate */
};
...
```

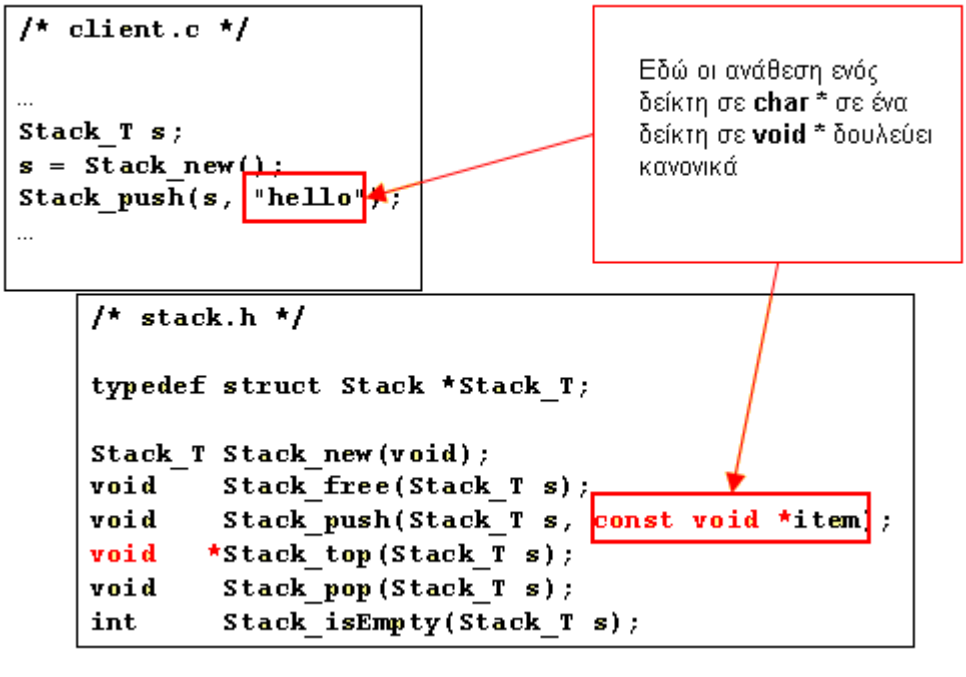
```
Stack_T s;
struct Item item;
item.str = "hello";
s = Stack_new();
Stack_push(s, item);
...
```

Με αυτό τον τρόπο όμως μπορούν να προκύψουν διάφορα προβλήματα. Καταρχάς θα πρέπει πάντοτε το κάθε πρόγραμμα πελάτης να ορίζει τον τύπο των δεδομένων που θα χρησιμοποιεί η δομή. Με αυτό τον τρόπο θα επωμίζεται και το επιπλέον κόστος ελέγχου και συντήρησης της υλοποίησης του. Επίσης το πρόγραμμα πελάτης ενδέχεται να χρειάζεται 2 διαφορετικές στοίβες για διαφορετικούς τύπους δεδομένων και με αυτήν την υλοποίηση δεν είναι εφικτό.

Μια άλλη λύση είναι να ορίσουμε τον τύπο του στοιχείου της στοίβας σαν **void ***

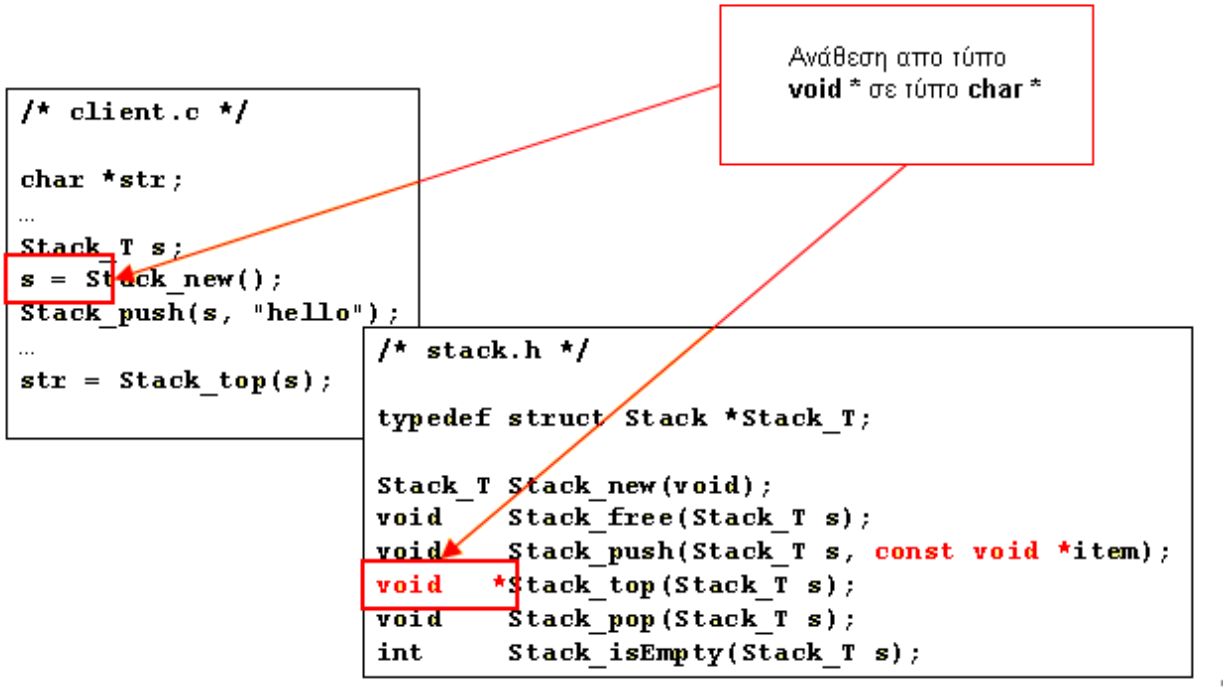
```
/* stack.h */
typedef struct Stack *Stack_T;
Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
void    Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Με αυτό τον τρόπο μπορούμε να αναθέτουμε δείκτες από οποιοδήποτε τύπο στο δείκτη σε void.



Σχήμα 13: Ανάθεση δείκτη σε void.

Επίσης είναι σωστό να αναθέτουμε την τιμή που επιστρέφει η συνάρτηση top (void *) σε μια μεταβλητή τύπου δείκτη σε χαρακτήρα (char *).



Σχήμα 14: Ανάθεση επιστρεφόμενης τιμής σε void *

Με τη συγκεκριμένη υλοποίηση όμως προκύπτει ένα σημαντικό πρόβλημα. Το πρόγραμμα πελάτης πρέπει να γνωρίζει κάθε φόρα από πριν τον πραγματικό τύπο των δεδομένων που αναφέρεται ένας δείκτης void.

Έστω το εξής παράδειγμα :

```
/* client.c */
int *i;
…
Stack_T s;
s = Stack_new();
Stack_push(s, "hello");
…
i = Stack_top(s);
```

Εδώ εισάγουμε στη στοίβα μια συμβολοσειρά “hello” και στη συνέχεια προσπαθούμε να εξάγουμε αυτή την τιμή σε ένα δείκτη για ακέραιους. Τυπικά δεν θα προκύψει κανένα λάθος στην μεταγλώττιση του προγράμματος καθώς αυτό που κάνουμε είναι επιτρεπτό. Στην πραγματικότητα όμως θα έχουμε πρόβλημα να διαβάσουμε τα δεδομένα της στοίβας εφόσον χρησιμοποιούμε λάθος τύπο. Γενικά η χρήση δεικτών void παρεμποδίζει την σωστή λειτουργία του μεταγλωττιστή για τον έλεγχο του προγράμματος. Το συγκεκριμένο πρόβλημα δεν έχει λύση. Θα πρέπει να είναι υπεύθυνος ο προγραμματιστής για τον έλεγχο τυχόν ασυμφωνιών σε αυτούς τους τύπους.

Ένα άλλο πρόβλημα που μπορεί να προκύψει με τη χρήση δεικτών void έχει πάλι να κάνει με τα δεδομένα που εισάγουμε. Για να αποθηκεύσουμε μια ακέραια τιμή, για παράδειγμα, θα μπορούσαμε να το γράψουμε ως εξής:

```
Stack_push(s, 5);
```

Αυτό όμως είναι λάθος καθώς η συνάρτηση μας δέχεται σαν ορίσματα μόνο δείκτες. Οπότε δεν μπορούμε να χρησιμοποιήσουμε απ' ευθείας απλούς τύπους δεδομένων (int, float κ.τ.λ.π.). Ο σωστός τρόπος είναι :

```
int i = 5;

Stack_push(s, &i);
```

Η συγκεκριμένη λύση είναι σωστή μέν, κάνει όμως τον κώδικά μας πιο πολύπλοκο και επιρρεπή σε λάθη.

Γενικά σε μια γλώσσα προγραμματισμού όπως είναι η C δεν υπάρχει κάποιος μηχανισμός για να δημιουργήσουμε πραγματικά generic δομές. Στην C++ αυτό επιτυγχάνεται με τη χρήση πρότυπων (template) κλάσεων και συναρτήσεων ενώ στη Java έχουμε generic κλάσεις.

8.2 Παραδείγματα Generic Αλγορίθμων

Έστω ότι θέλουμε να προσθέσουμε μια ακόμα συνάρτηση στο δομικό μας στοιχείο στοίβας:

```
/* stack.h */
typedef struct Stack *Stack_T;
Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
void    Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2);
```

Η συνάρτηση αυτή θα χρησιμοποιείτε για να ελέγχει αν 2 στοίβες είναι ίσες. Θα επιστρέφει 1 όταν οι 2 στοίβες έχουν στοιχεία ίσα αναμεταξύ τους και στην ίδια σειρά.

Μια πρώτη προσπάθεια για την υλοποίηση αυτής της συνάρτησης είναι η παρακάτω.

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
  return s1 == s2;
}
```

Η συγκεκριμένη υλοποίηση έχει ένα πρόβλημα όμως. Αυτό που ελέγχουμε πραγματικά εδώ είναι αν οι δείκτες της κάθε στοίβας s1 και s2 είναι ίδιοι. Αν δηλαδή και οι 2 δείκτες αναφέρονται στην ίδια θέση μνήμης. Ο επιθυμητός σκοπός αυτής της συνάρτησης είναι διαφορετικός, επειδή θέλουμε να ελέγξουμε αν 2 διαφορετικές στοίβες έχουν ίσα στοιχεία.

Θα βοηθούσε ίσως να εξηγήσουμε καλύτερα το θέμα που προκύπτει με τον τελεστή ==. Έστω ότι έχουμε 2 θέσεις στη μνήμη που περιέχουν τις ίδιες τιμές.

```
int i = 5;
int j = 5;
```

Οι διευθύνσεις αυτές είναι διαφορετικές όμως. Οπότε και ο έλεγχος (&i == &j) επιστρέφει λάθος. Για να συγκρίνουμε τις μεταβλητές αυτές θα πρέπει να συγκρίνουμε τις τιμές τους, i == j. Αυτού του είδους η σύγκριση όμως λειτουργεί για συγκεκριμένους τύπους δεδομένων. Αν θέλαμε να συγκρίνουμε 2 συμβολοσειρές, για παράδειγμα:

```
char * s1;
```

```
char * s2;
```

χρησιμοποιώντας τον έλεγχο s1 == s2 θα καταλήγαμε να συγκρίνουμε τις διευθύνσεις τους και όχι τα πραγματικά δεδομένα. Ο σωστός τρόπος θα ήταν με τη χρήση της συνάρτησης strcmp().

```
/* stack.c */
…
```

Στο παρακάτω παράδειγμα βλέπουμε την επόμενη προσπάθεια μας για την υλοποίηση της συνάρτησης. Εδώ χρησιμοποιούμε μια επαναληπτική διαδικασία για να προσπελάσουμε όλα τα στοιχεία της κάθε στοίβας και να τα ελέγξουμε ένα – ένα :

```
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;

    while ((p1 != NULL) && (p2 != NULL)) {
        if (p1 != p2) {
            return 0;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL)) {
        return 0;
        }
    return 1;
}
```

Η επαναληπτική λογική που χρησιμοποιούμε εδώ είναι σωστή, εξακολουθούμε όμως να επαναλαμβάνουμε το ίδιο λάθος με πριν. Αυτό που συγκρίνουμε εδώ είναι οι δείκτες για τα στοιχεία τις στοίβας και όχι τα ίδια τα στοιχεία

Μπορούμε να βελτιώσουμε την συνθήκη για τη σύγκριση που κάνουμε ως εξής :

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;

    while ((p1 != NULL) && (p2 != NULL)) {
        if (p1->item != p2->item) {
            return 0;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL)) {
        return 0;
        }
    return 1;
}
```

Εδώ όμως παρατηρούμε ότι ο έλεγχος που κάνουμε είναι πάλι για τους δείκτες στα στοιχεία της στοίβας και όχι στα δεδομένα τους. Οπότε ούτε και αυτή η λύση είναι σωστή.

Για να μπορέσουμε να ελέγξουμε τις τιμές αυτών των δεικτών θα πρέπει να χρησιμοποιήσουμε τη συνάρτηση strcmp(). Το παράδειγμα μας τώρα γίνεται ως εξής :

```
/* stack.c */
…
int Stack_areEqual(Stack_T s1, Stack_T s2) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;

    while ((p1 != NULL) && (p2 != NULL)) {
        if (strcmp(p1->item, p2->item) != 0){
            return 0;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL)){
        return 0;
    }
    return 1;
}
```

Η υλοποίηση αυτή θα μας επιστρέψει το επιθυμητό αποτέλεσμα. Είναι προφανές όμως ότι λειτουργεί σωστά μόνο με δεδομένα τύπου συμβολοσειράς (strings). Εφόσον προσπαθούμε να κατασκευάσουμε μια υλοποίηση που θα δουλεύει ανεξαρτήτως τύπου, θα είναι δηλαδή πραγματικά generic, τότε θα πρέπει να απορρίψουμε και αυτή την λύση.

Μετά από όλες τις παραπάνω αποτυχημένες προσπάθειες καταλήγουμε στο ότι η ιδανική λύση θα ήταν να ορίσουμε μια συνάρτηση σύγκρισης των στοιχείων η οποία θα υλοποιείται από το πρόγραμμα του πελάτη. Θα πρέπει στην δήλωση της συνάρτησης μας **Stack_areEqual()** να χρησιμοποιήσουμε μια επιπλέον παράμετρο, την συνάρτηση σύγκρισης μας.

```
/* stack.h */
typedef struct Stack *Stack_T;
Stack_T Stack_new(void);
void    Stack_free(Stack_T s);
void    Stack_push(Stack_T s, const void *item);
void   *Stack_top(Stack_T s);
void    Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
int     Stack_areEqual(Stack_T s1, Stack_T s2, int (*cmp)(const void *item1,
                                                          const void      *item2));
```

Η συνάρτηση αυτή θα δέχεται ως παραμέτρους 2 δείκτες void (ένα για το κάθε στοιχείο που θέλουμε να συγκρίνουμε) και θα επιστρέφει μια ακέραια τιμή (0 ή 1) ανάλογα με το αποτέλεσμα. Η υλοποίηση μας τώρα γίνεται ως εξής:

```
/* stack.c */
…
```

```
int Stack_areEqual(Stack_T s1, Stack_T s2,
                   int (*cmp)(const void *item1, const void *item2)) {
    struct Node *p1 = s1->first;
    struct Node *p2 = s2->first;

    while ((p1 != NULL) && (p2 != NULL)) {
        if ((*cmp)(p1->item, p2->item) != 0) {
            return 0;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
    if ((p1 != NULL) || (p2 != NULL)) {
        return 0;
    }
    return 1;
}
```

Το πλεονέκτημα εδώ λοιπόν είναι ότι επιτρέπουμε στο πρόγραμμα πελάτη να δηλώσει τη δικιά του συνάρτηση σύγκρισης δεδομένου ότι γνωρίζει τον τύπο των δεδομένων που θέλει να χρησιμοποιήσει. Ένα παράδειγμα τέτοιας συνάρτησης για συμβολοσειρές θα μπορούσε να είναι η εξής:

```
int strCompare(const void *item1, const void *item2) {
    char *str1 = item1;
    char *str2 = item2;

    return strcmp(str1, str2);
}
```

Και για ακεραίους :

```
int numCompare (const void *item1, const void *item2) {
    if( *(int*)item1 == *(int*)item2 ) {
        return 1;
    }
    return 0;
}
```

Στο πρόγραμμα πελάτης θα την χρησιμοποιήσουμε ως εξής:

```
char str2[] = "hi";
Stack_T s1 = Stack_new();
Stack_T s2 = Stack_new();
Stack_push(s1, str1);
Stack_push(s2, str2);
if (Stack_areEqual(s1,s2,strCompare)) {
…
}
```

Εδώ περνάμε σαν παράμετρο την διεύθυνση της συνάρτησης **strCompare** η οποία έχει ακριβώς τις ίδιες παραμέτρους με την συνάρτηση που δέχεται ως όρισμα η **Stack_areEqual().**

Ειδικά για την περίπτωση ελέγχου συμβολοσειρών θα μπορούσαμε να χρησιμοποιήσουμε απ' ευθείας τη συνάρτηση **strcmp()** αρκεί να κάναμε την απαραίτητη μετατροπή στους τύπους των παραμέτρων:

```
Stack_areEqual(s1, s2, (int (*)(const void*, const void*))strcmp)
```

Στο παρακάτω σχήμα κάνουμε μια σύντομη περίληψη σε όσα αναφέρθηκαν στα 2 προηγούμενα κεφάλαια (Modularity και Generics).

Πίνακας 3: Υλοποίηση Ενοτήτων (Modules) στην C και Γενικότητα (Genericity) [43].

		Modularity				
		No Modularity (0)	Απόκρυψη Πράξεων (ΜΑ) (1)	Απόκρυψη Δεδομένων και Πράξεων (ΠΑ)(2,3)	+	-
Genericity	ΤΣ καθορισμένος στον κώδικα, χωρίς αφαίρεση	ΟΧΙ (Demo Mod 0)	ΟΧΙ λόγω G (Demo Mod 1)	ΟΧΙ λόγω G (Demo Mod 3+2)	έλεγχος μεταγλώττισης	1 μόνο ΤΣ, Αλλαγή ΤΣ απαιτεί Αλλαγή Υλοποίησης ΑΤΔ, απαιτεί recompilation
	ΤΣ #include σε .h (και αντίστοιχο .c)	ΟΧΙ λόγω Modularity	ΝΑΙ με Προσοχή όπως ουρά 1ης άσκησης	ΝΑΙ (απλό με πολλά πλεονεκτήματα) ουρά 2ης άσκησης	Αλλαγή ΤΣ δεν απαιτεί αλλαγή υλοποίησης ΑΤΔ, έλεγχος μεταγλώττισης	1 μόνο ΤΣ, Αλλαγή ΤΑ απαιτεί recompilation
	ΤΣ (Πλήρης απόκρυψη και ΤΣ *Τ) (παράμετροι συναρτήσεων και data του ΑΤΔ είναι *Τ)	ΟΧΙ λόγω Modularity	Χωρίς ιδιαίτερο όφελος	ΝΑΙ, απαιτεί καλή χρήση δεικτών Mod 3 Gen 1-T (SetValue απλουστεύεται) (compare)	Αλλαγή ΤΣ δεν απαιτεί αλλαγή υλοποίησης ΑΤΔ, δεν απαιτεί recompilation, έλεγχος μεταγλώττισης	1 μόνο ΤΣ
	ΤΣ (Πλήρης απόκρυψη ΤΣ *Τ)(παράμετροι συναρτήσεων και data του ΑΤΔ είναι void*)	ΟΧΙ λόγω Modularity	Χωρίς ιδιαίτερο όφελος	Χρήσιμη για βιβλιοθήκες, απαιτεί καλούς ελέγχους. Mod 3 Gen n-Type (compare)	πολλά-ΤΣ, σταθερή υλοποίηση ΑΤΔ, δεν απαιτεί recompilation	χωρίς έλεγχο μεταγλώττισης
	+		Καλή προσέγγιση, προσοχή στην άμεση χρήση στον Τύπο.	Η καλύτερη απόκρυψη		
	-	Να αποφεύγεται εντελώς, καμία απόκρυψη	Κίνδυνοι άμεσης πρόσβασης στον ΤΑ, αλλοίωση απόκρυψης	Διαχειριζόμαστε δείκτες και όχι structs		

9. ΣΥΜΠΕΡΑΣΜΑΤΑ

Έχοντας ακολουθήσει τις οδηγίες που αναφέρονται στα προηγούμενα κεφάλαια μπορούμε να καταλήξουμε στο συμπέρασμα ότι η ποιότητα του κώδικα μας έχει βελτιωθεί αισθητά. Μπορούμε να παρατηρήσουμε αυτή τη βελτίωση και στη μείωση του χρόνου αποσφαλμάτωσης και επίλυσης λαθών στα προγράμματα μας. Ο πηγαίος μας κώδικας πλέον είναι πιο ευανάγνωστος και πιο εύκολα κατανοητός από τους συναδέλφους μας. Επίσης είναι πιο εύκολο να συνεργαστούμε με άλλους προγραμματιστές σε μεγαλύτερα και πιο πολύπλοκα προγράμματα.

Οι τεχνικές αυτές προγραμματισμού δεν περιορίζονται μόνο στις τρέχουσες συνθήκες που επικρατούν στα συστήματα και τις γλώσσες προγραμματισμού. Θα παρατηρήσουμε στο μέλλον ότι εξακολουθούν να έχουν αξία και όταν τις χρησιμοποιήσουμε με καινούργιες τεχνολογίες. Οι έννοιες της απλότητας, διαφάνειας και γενικότητας στον κώδικα μας θα είναι ένα χρήσιμο εργαλείο σε οποιαδήποτε προγραμματιστική γλώσσα επιλέξουμε.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Programming style	Μορφή προγραμματισμού
Variables	Μεταβλητές
Functions	Συναρτήσεις
Pointers	Δείκτες
Comments	Σχόλια
Indentation	Οριζόντια Διαστήματα
Nested Structures	Εμφωλευμένες δομές
Robustness	Ανθεκτικότητα
Echo printing	Άμεση εκτύπωση την δεδομένων εισόδου
Run time errors	Λάθοι κατά την εκτέλεση
Shift	Μετατόπιση
Record	Εγγραφή
End of file	Τέλος του αρχείου
By value	Κατ' αξία
Signal flag	Σημαίνουσες σημαίες
Portability	Μεταφορά προγράμματος
Standard	Βασική έκδοση
Source code	Πρωτογενής κώδικας
Conditional compilation	Υπό συνθήκη μεταγλώττιση
Interface	Διεπαφή
Bug	Σφάλμα
Abstraction	Αφαίρεση
Enums	Απαριθμήσεις
Compile	Μεταγλώττιση
Absraction data type	Αφηρημένος τύπος δεδομένων
ADT	ΑΤΔ
Allocate	Εκχωρήσουν
Input/Output	Εισόδου/Εξόδου
Single entry single exit	Μονής εισόδου μονής εξόδου
Module	Δομικό στοιχείο
Debugging	Αποσφαλμάτωση
Testing	Έλεγχος
Top down design	Σχεδίαση από το γενικό προς το ειδικό
Stepwise refinement	Εκλεπτυσμός με διαδοχικά βήματα
Treelike structure	Δενδροειδή δομή
Subtasks	Υποστόχοι
Bottom up method	Από την ειδική προς την γενική μέθοδος
Program unit	Μονάδα προγράμματος
Regular expressions	Κανονικές εκφράσεις
Boundaries	Οριακές περιπτώσεις
Statement testing	Έλεγχος εντολών
Path testing	Έλεγχος πιθανών ροών εκτέλεσης
Binary tree	Δυαδικό δέντρο
Stress Testing	Έλεγχος καταπόνησης
Assertions	Συμβάσεις-Συμφωνίες
Framework	Πλαίσιο
Buffer zone	Ζώνη προστασίας

Unit testing	Ο έλεγχος μιας συνάρτησης
Feature	Χαρακτηριστικό
Integration Testing	Έλεγχοι ενσωμάτωσης
Acceptance Testing	Έλεγχος παραδεχόμενων
Program Modularity	Διάσπαση προγράμματος σε δομικά στοιχεία
Generic Modules	Γενικού τύπου δομικά στοιχεία
Strings	Συμβολοσειρές
Encapsulation	Ενθυλάκωση
Memory leaks	Διαρροή μνήμης
Segmentation faults	Σφάλματα κατάμτησης
File descriptors	Περιγραφείς αρχείων
Stack	Στοίβα
Null	Κενός
Template	Πρότυπο
Coupling	Ζύζευξη
External testing	Εξωτερικός έλεγχος
Internal testing	Εσωτερικός έλεγχος
Automation	Αυτοματισμοί

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

π.χ.	Παραδείγματος χάριν
EOF	End of file
ΑΤΔ	Αφαιρετικός τύπος δεδομένων
ADT	Absraction Data Type
I/O	Input/Output
κ.τ.λ.π	και τα λοιπά
εφ	εφαπτομένη

ΑΝΑΦΟΡΕΣ

[1] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 7.
[2] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 3.
[3] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 3.
[4] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 4.
[5] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 24.
[6] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 25.
[7] Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, pp. 27-28.
[8] Brian W. Kernighan, Dennis M. Ritchie, Η Γλώσσα Προγραμματισμού C, Prentice Hall Software Series, Κλειδάριθμος 2005, pp. 48-49.
[9] Brian W. Kernighan, Dennis M. Ritchie, Η Γλώσσα Προγραμματισμού C, Prentice Hall Software Series, Κλειδάριθμος 2005, p. 49.
[10]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 63.
[11]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 65.
[12]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 65.
[13]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 66.
[14]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 190.
[15]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 196.
[16]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 198.
[17]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 199.
[18]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 202.
[19]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, pp. 204-205.
[20]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 211.
[21]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 210.
[22]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 192.
[23]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 2.
[24]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 3.
[25]J. Jagger, C Programming – Advanced Techmiques, 2016, pp. 5-7.
[26]J. Jagger, C Programming – Advanced Techmiques, 2016, pp. 9-10.
[27]J. Jagger, C Programming – Advanced Techmiques, 2016, pp. 11-12.
[28]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 13.
[29]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 14.
[30]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 19.
[31]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 20.
[32]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 24.
[33]J. Jagger, C Programming – Advanced Techmiques, 2016, p. 25.
[34]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, pp. 47-49.
[35]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, pp. 50-51.
[36]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, pp. 52-53.
[37]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, pp. 54-55.

[38]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 56.
[39]Π. Σταματόπουλος, Σημειώσεις Εισαγωγής Στον Προγραμματισμό, 2016, p. 57.
[40]Brian W. Kernighan, Rob Pike, The Practise of Programming, Addison-Wesley, 1999, p. 139.
[41]Ι. Κοτρώνης, Σημειώσεις Μαθηματος Δομές Δεδομένων και Τεχνικές Προγραμματισμού, ΕΝΟΤΗΤΑ 5 - ΤΕΧΝΙΚΕΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ, p. 19; http://opencourses.uoa.gr/modules/document/?course=DI105
[42]Ι. Κοτρώνης, Σημειώσεις Μαθηματος Δομές Δεδομένων και Τεχνικές Προγραμματισμού, κεφάλαιο 9 (Testing Revised), p. 5; https://eclass.uoa.gr/modules/units/?course=D419&id=12190
[43]Ι. Κοτρώνης, Σημειώσεις Μαθηματος Δομές Δεδομένων και Τεχνικές Προγραμματισμού, Φροντιστήριο Generics, p. 24; https://eclass.uoa.gr/modules/units/?course=D419&id=12190