ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ «ΠΛΗΡΟΦΟΡΙΚΗ»

Στυλ προγραμματισμού

Αναγνωστόπουλος Βασίλης - Θάνος

ΑΘΗΝΑ, 2014

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

1. Εισαγωγή

Στυλ προγραμματισμού (αγγλ. Programming/Coding style ή Coding conventions) είναι ένα σύνολο κανόνων ή κατευθυντήριων γραμμών που χρησιμοποιούνται κατά την σύνταξη του πηγαίου κώδικα ενός προγράμματος υπολογιστή σε μια συγκεκριμένη γλώσσα προγραμματισμού που συνιστά να χρησιμοποιούνται συγκεκριμένες πρακτικές και μέθοδοι για κάθε πτυχή ενός προγράμματος γραμμένο στην συγκεκριμένη γλώσσα. Ο σκοπός του προγραμματιστικού στυλ είναι να εξασφαλίσει ότι ο πηγαίος κώδικας είναι γραμμένος με τρόπο που βελτιώνει την αναγνωσιμότητα του καθώς και την συντηρησιμότητα του [30, 31, 1, 19, 20].

Το σύνολο των κανόνων που ισχύουν για την συγγραφή του πηγαίου κώδικα μπορούν να ταξινομηθούν σε 4 τομείς [19] (βλ. σχήμα 1.1):

Τυπογραφικά στυλ: Κανόνες που αφορούν την διάταξη και την εμφάνιση του κώδικα καθώς και την χρήση σχολίων, αλλά όχι την εκτέλεση του προγράμματος.

Γενικές πρακτικές προγραμματισμού: Κανόνες και κατευθυντήριες γραμμές σχετικά με την μεθοδολογία και την γλώσσα που χρησιμοποιούνται και επηρεάζουν τον πηγαίο κώδικα.

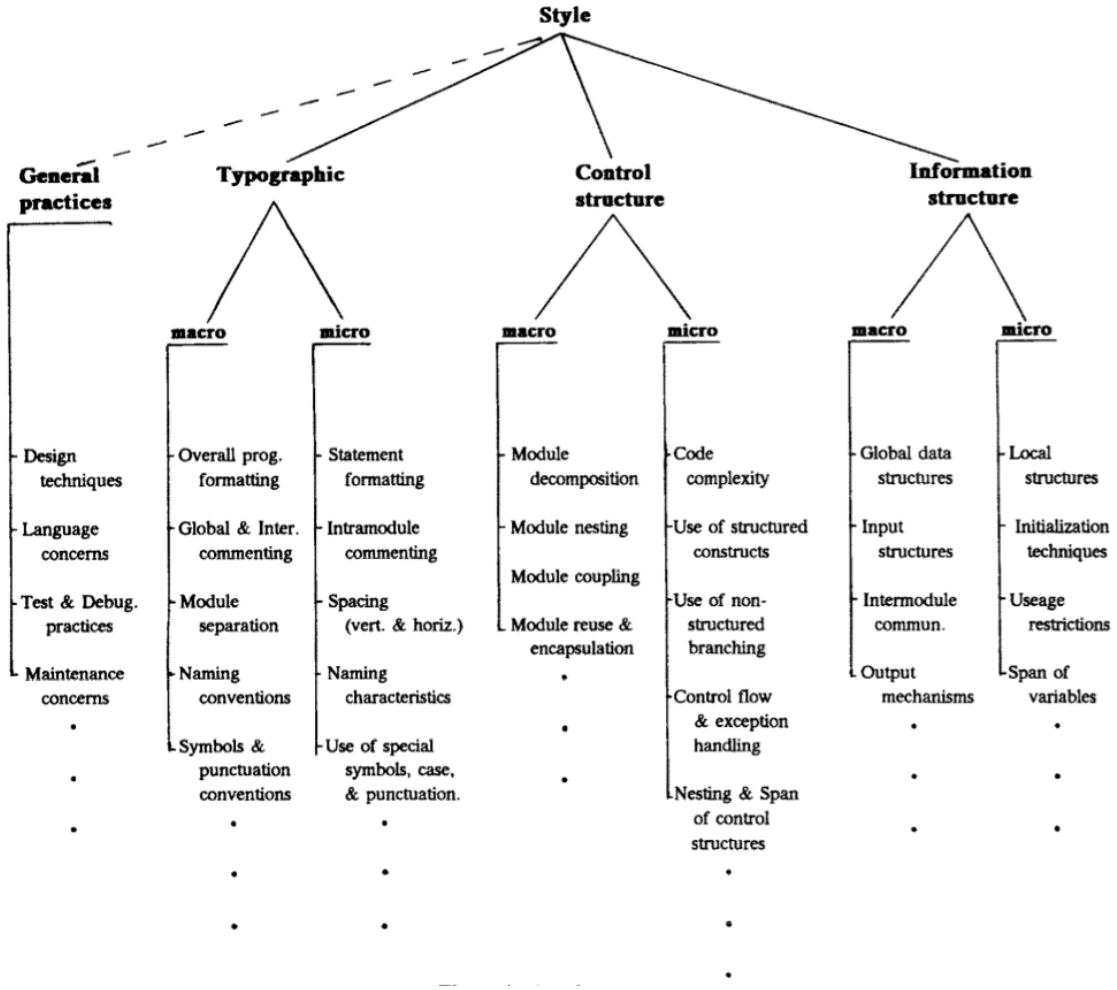
Δομές ελέγχου: Κανόνες που επηρεάζουν την χρήση των αλγορίθμων καθώς και την υλοποίηση τους.

Δομές πληροφοριών : Κανόνες που επηρεάζουν τις δομές δεδομένων, την ροή του προγράμματος καθώς και την αποθήκευση και χειρισμό της πληροφορίας.

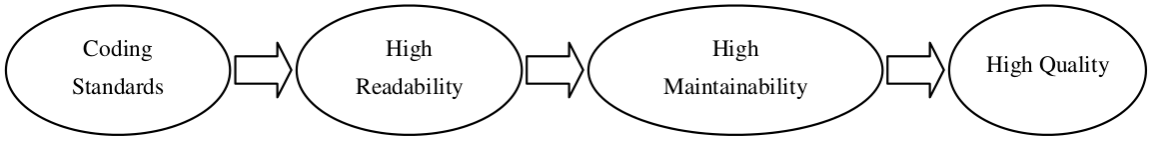
Σε αυτή την εργασία θα αναπτυχθούν τα δύο πρώτα θέματα (τυπογραφικό στυλ και γενικές πρακτικές προγραμματισμού). Αυτές οι κατευθυντήριες γραμμές είναι για την διαρθρωτική ποιότητα του λογισμικού[1] (αγγλ. Software structural quality). Προβάλλεται συχνά το επιχείρημα ότι ακολουθώντας ένα συγκεκριμένο προγραμματιστικό στυλ μπορεί να βοηθήσει στην ανάγνωση και κατανόηση του πηγαίου κώδικα, και επιπλέον βοηθά στην αποφυγή λαθών αλλά και στην βελτίωση της ποιότητας του πηγαίου κώδικα [1, 2, 5, 26] (βλέπε ακόμα σχ. 1.2). Οι συμβάσεις που γίνονται όταν ακολουθείται ένα συγκεκριμένο στυλ προγραμματισμού βοηθά μόνο τους συντηρητές και τους διορθωτές του πηγαίου κώδικα ενός προγράμματος. Το προγραμματιστικό στυλ δεν επιβάλλεται από τους μεταγλωττιστές. Έτσι ένα πρόγραμμα το οποίο δεν ακολουθεί κάποιους ή όλους τους κανόνες δεν έχει καμία επίδραση στο εκτελέσιμο αρχείο του προγράμματος που δημιουργήθηκε από τον πηγαίο κώδικα [30, 31].

Το στυλ γραφής είναι ένα στοιχεία το οποίο συχνά παραβλέπεται αλλά είναι πολύ κρίσιμο χαρακτηριστικό της γραφής. Το ύφος της γραφής επηρεάζει άμεσα την αναγνωρι-

[1] Η διαρθρωτική ποιότητα του λογισμικού αναφέρεται στο πως οι μη – λειτουργικές πτυχές του προγραμματισμού (όπως τα σχόλια, κ.λ.π) βοηθούν στην υλοποίηση των λειτουργικών πτυχών, όπως η αξιοπιστία του προγράμματος ή ευκολία στην συντήρηση του [32].



Σχήμα 1.1: Η ταξινόμηση του προγραμματιστικού στυλ [21].



Σχήμα 1.2: Η σχέση μεταξύ προγραμματιστικού στυλ και ποιότητας λογισμικού [26].

σιμότητα και κατανόησης του τελικού προϊόντος. Έτσι ομοίως και το στυλ προγραμματισμού, που είναι η συγγραφή του πηγαίου κώδικα σε μία γλώσσα προγραμματισμού, ομοίως πάσχει από αυτή την παραμέληση. Τα προγράμματα πρέπει να είναι αναγνώσιμα και κατανοητά όχι μόνο από τις μηχανές αλλά ομοίως και από τον άνθρωπο. Η απαίτηση αυτή είναι σημαντική για τη δημιουργία ποιοτικών προϊόντων που ανταποκρίνονται όχι μόνο στις ανάγκες των χρηστών, αλλά επίσης μπορούν να αναπτυχθούν εντός προγράμματος και εκτιμώμενου κόστους. Η αποτελεσματική συγγραφή προγραμμάτων έχει ήδη ερευνηθεί από πολύ παλιά ([15]) από τις πρώτες γλώσσες προγραμματισμού που υπήρχαν όπως η Fortan αλλά συνεχίζει να είναι ακόμα ένα φλέγον ζήτημα [1, 17].

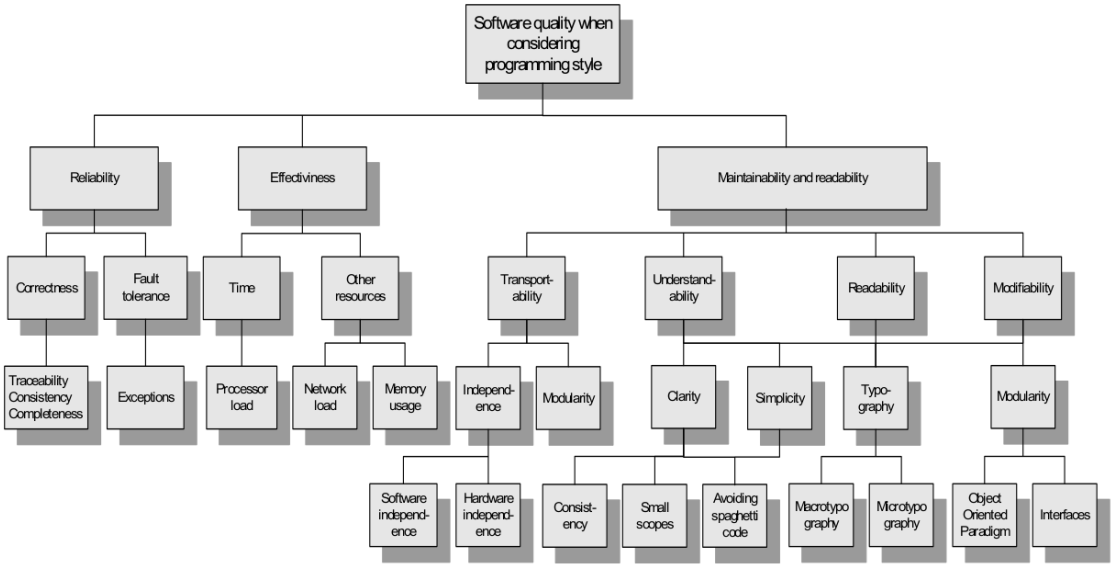
Το στυλ προγραμματισμού που χρησιμοποιείται σε ένα συγκεκριμένο πρόγραμμα μπορεί να προέρχεται από τις συμβάσεις γραφής (αγγλ. coding conventions) που έχει κάνει μια εταιρεία ή κάποιος άλλος οργανισμός που ασχολείται με την συγγραφή κώδικά (π.χ. [7, 10, 12, 11, 8, 3, 25]) καθώς και τις προτιμήσεις του ίδιου του συγγραφέα. Υπάρχουν πολυάριθμες συμβάσεις γραφής που χρησιμοποιούνται για να διασφαλίσουν "συνεπή" (αγλλ. consistent) κώδικα. Οι συμβάσεις γραφής είναι σημαντικές στους προγραμματιστές διότι μπορούν να προσφέρουν αρκετά πλεονεκτήματα όπως [27, 30, 24]:

Συντήρηση λογισμικού (αγγλ. Software maintenance): Η συντήρηση του λογισμικού αντιπροσωπεύει τουλάχιστον το 50% του κόστους ζωής του λογισμικού[19], ενώ σε άλλες έρευνες αναφέρετε ότι το 40% με 80% τους κόστους ζωής ενός λογισμικού πάει στην συντήρηση του κώδικα [9, 30]. Η μείωση, λοιπόν, του κόστους συντήρησης είναι ο πιο συχνά αναφερόμενος λόγος για την χρησιμοποίηση συμβάσεων γραφής. Σχεδόν κανένα λογισμικό δεν διατηρείται μόνο από τον αρχικό του δημιουργό. Έτσι οι συμβάσεις αυτές βοηθούν στην καλύτερη κατανόηση του κώδικα, επιτρέποντας και σε άλλους προγραμματιστές να συμμετέχουν στην επέκταση του κώδικα.

Ποιότητα λογισμικού (αγγλ. Software quality): Η αξιολόγηση και η διόρθωση του κώδικα συχνά περιλαμβάνει την ανάγνωση του πηγαίου κώδικα από τρίτους. Εξ ορισμού, μόνο ο αρχικός συντάκτης έχει διαβάσει τον πηγαίο κώδικα πριν την αξιολόγηση του. Ο πηγαίος κώδικας ο οποίος τηρεί κάποιες συμβάσεις είναι πιο κατανοητός στους υπόλοιπους και επομένως η διαδικασία ανίχνευσής σφαλμάτων γίνεται πιο εύκολη.

Μειώνουν την πολυπλοκότητα: Όσο πιο σύνθετο είναι ένα έργο λογισμικού τόσο πιο πιθανό είναι να έχει σφάλματα. Οι σωστές συμβάσεις γραφής βοηθούν στην εύρεση αυτών των σφαλμάτων και στην μείωση της πολυπλοκότητας του πηγαίου κώδικα.

Βελτιωμένη ταχύτητα ανάπτυξης και καλύτερη ομαδική εργασία: Οι προγραμματιστές δεν χρειάζεται να ξεκινούν πάντα από το μηδέν και έχουν μία στέρεα βάση για την ανάπτυξη του λογισμικού, ενώ ταυτόχρονα οι συμβάσεις συμβάλλουν στην μείωση των περιττών συζητήσεων σχετικά με ασήμαντα θέματα και διευκολύνουν



Σχήμα 1.3: Τα πεδία που επηρεάζονται από την επιλογή ενός στυλ κώδικα [1].

στην συνεργασία μεταξύ των προγραμματιστών.

Τέλος τα πεδία που επηρεάζονται από την επιλογή ενός στυλ κώδικα φαίνονται συνοπτικά στο σχήμα 1.3.

2. Στοιχεία καλού προγραμματιστικού στυλ

Το καλό στυλ στην συγγραφή κώδικα είναι υποκειμενικό θέμα και είναι δύσκολο να προσδιοριστεί. Παρόλα αυτά, υπάρχουν πολλά στοιχεία που είναι κοινά σε ένα μεγάλο αριθμό από στυλ προγραμματισμού. Σε αυτή την ενότητα θα παρατεθούν αυτά ακριβώς τα στοιχεία.

2.1 Τυπογραφικό στυλ - Εμφάνιση κώδικα

Το προγραμματιστικό στυλ συνήθως ασχολείται και με την οπτική εμφάνιση του πηγαίου κώδικα, με στόχο να απαιτείται λιγότερη προσπάθεια από τους προγραμματιστές για την αναγνώριση και την εξαγωγή χρήσιμων πληροφοριών από τον κώδικα . Όταν το πρόγραμμα ολοκληρωθεί, σπάνια ένας προγραμματιστής θα το διαβάσει από πάνω προς τα κάτω. Στον εντοπισμό σφαλμάτων και στην βελτιστοποίηση του προγράμματος, οι προγραμματιστές παραλείπουν συχνά μεγάλα τμήματα του κώδικα, προκειμένου να βρουν αυτό που ψάχνουν. Μια καλή αναλογία είναι αν συγκριθεί ο πηγαίος κώδικας με ένα λεξικό. Αν οι λέξεις σε ένα λεξικό δεν ήταν σε αλφαβητική σειρά και με έντονη γραφή τότε ο εντοπισμός μιας συγκεκριμένης λέξης θα ήταν πολύ δύσκολος. Ομοίως λοιπόν και στον πηγαίο κώδικα, η οπτική εμφάνιση του κώδικα βοηθά στην μετέπειτα επεξεργασία του [31, 4].

Το τυπογραφικό στυλ έχει αποδειχθεί ότι επηρεάζει την δυνατότητα κατανόησης του πηγαίου κώδικα και η συνεπής εφαρμογή κάποιου τυπογραφικού στυλ ενισχύει την κατανόηση ενός προγράμματος [19].

2.1.1 Εσοχές του κώδικα

Οι εσοχές στον κώδικα βοηθούν στον προσδιορισμό της ροής του προγράμματος. Συγκεκριμένα οι εσοχές χρησιμοποιούνται για να επιτρέψουν στον αναγνώστη του κώδικα να καθορίσει το επίπεδο ένθεσης μίας δήλωσης με μία ματιά και για να οριοθετήσουν λογικά μπλοκ κώδικα. Για να είναι χρήσιμες, οι εσοχές πρέπει να είναι συνεπής και να χρησιμοποιείται πάντα ο ίδιος αριθμός σε όλο το πρόγραμμα (ή τουλάχιστον στο ίδιο αρχείο του προγράμματος). Συνήθως αναφέρονται ότι οι εσοχές πρέπει να είναι μεταξύ 2 και 5 κενών [31, 4, 24]. Παρακάτω ακουλουθούν δύο παραδείγματα που στο ένα χρησιμοποιούνται εσοχές ενώ στο άλλο όχι:

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
```

```
    return false;
}
```

Πρόγραμμα 1: Παράδειγμα εσοχών (1)

και

```
if  ( hours   < 24
   && minutes < 60
   && seconds < 60
)
{return    true
;}         else
{return   false
;}
```

Πρόγραμμα 2: Παράδειγμα εσοχών (2)

Το πρώτο παράδειγμα είναι πιο εύκολο να διαβαστεί μιας και κάθε λογικό κομμάτι του κώδικα ξεχωρίζει. Επομένως οι εσοχές στον κώδικα βοηθούν στις εμφολιασμένες εντολές.

2.1.2 Κατακόρυφη στοίχιση

Είναι συχνά χρήσιμο να στοιχίζονται κατακόρυφα παρόμοια στοιχεία, για να φαίνονται τα τυπογραφικά σφάλματα [31]. Για παράδειγμα:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');
```

Πρόγραμμα 3: Κατακόρυφη στοίχιση (1)

και

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');
```

Πρόγραμμα 4: Κατακόρυφη στοίχιση (2)

Το τελευταίο παράδειγμα κάνει δύο πράγματα διαισθητικά σαφές που πριν δεν ήταν:

- οι μεταβλητές search και replacement σχετίζονται μεταξύ τους
- και υπάρχει ένας παραπάνω όρος στην μεταβλητή search από ότι στην μεταβλητή replacement. Αν πρόκειται για κάποιο σφάλμα είναι πιο πιθανόν τώρα να εντοπιστεί.

Ωστόσο, σημειώνετε ότι υπάρχουν πολλά επιχειρήματα κατά της κατακόρυφης στοίχισης όπως:

Ευθραυστότητα: Εάν ένας προγραμματιστής κάνει κάποια αλλαγή στον ”πίνακα” και δεν τον τακτοποιήσει, έχει ως αποτέλεσμα την επιδείνωση της οπτικής εμφάνισης των στοιχείων, που γίνεται ακόμα χειρότερη με κάθε αλλαγή και

Δυσκολία στην συντήρηση: Η μορφοποίηση του πίνακα απαιτεί περισσότερη προσπάθεια για να διατηρηθεί.

2.1.3 Διαστήματα

Το στυλ που σχετίζονται με τα διαστήματα [2] χρησιμοποιούνται για την ενίσχυση της αναγνωσιμότητας του πηγαίου κώδικα. Δεν υπάρχουν γνωστές μελέτες οι οποίες υποστηρίζουν ότι τα διαστήματα βοηθούν στην αναγνωσιμότητα του κώδικα αλλά από μία απλή σύγκριση του παρακάτω κώδικά φαίνεται ότι αμυδρά βοηθά στην καλύτερη κατανόηση του κώδικα. Για παράδειγμα [31]:

```
int i;
for(i=0;i<10;++i){
    printf("%d",i*i+i);
}
```

Πρόγραμμα 5: Διαστήματα (1)

έναντι

```
int i;
for( i = 0; i < 10; ++i ) {
    printf( "%d", i * i + i );
}
```

Πρόγραμμα 6: Διαστήματα (2)

Η χρήση των διαστημάτων στον πηγαίο κώδικα είναι όμοια με τους κανόνες της αγγλικής γλώσσας. Αυτό σημαίνει ότι [4]:

1. Τα περισσότερα βασικά σύμβολα στις γλώσσες προγραμματισμού (π.χ. ”=”, ”+”, κ.λ.π.) θα πρέπει να έχουν τουλάχιστον ένα διάστημα πριν και ένα διάστημα μετά από αυτούς με τις παρακάτω εξαιρέσεις:

   (α) Δεν εμφανίζεται διάστημα πριν από κόμμα ή πριν από ερωτηματικό.

   (β) Δεν εμφανίζεται διάστημα πριν ή μετά από τελεία.

   (γ) Δεν εμφανίζεται διάστημα μεταξύ των δυαδικών τελεστών (π.χ. ”->”, ”++”).

2. Περισσότερα από ένα κενά μπορούν να χρησιμοποιηθούν για την ευθυγράμμιση στοιχείων (όπως στην κατακόρυφη στοίχιση).

3. Κενές γραμμές θα πρέπει ακόμα να χρησιμοποιούνται για να τον διαχωρισμό λογικών μπλοκ κώδικάς, όπως

[2] Τα κενά (διαστήματα), τα tabs και οι νέες γραμμές (αλλαγή γραμμής) ονομάζονται διαστήματα.

(α) Στο αρχή του πηγαίου κώδικα όπου υπάρχουν οι ντιρεκτίβες include, const, typedef κ.λ.π. .

(β) και σε κομμάτια κώδικα που είναι εκτεταμένα και υπάρχουν μέσα τους ξεχωριστά τμήματα κώδικα και μπορούν να διαχωριστούν με μία κενή γραμμή.

2.1.4 Αγκύλες

Στις γλώσσες προγραμματισμού που επιτρέπουν αγκύλες, έχει καταστεί κοινή πρακτική να χρησιμοποιούνται ακόμα και όταν η χρήση τους δεν είναι απαραίτητη. Η χρήση τους επιτρέπεται σε όλους τους βρόγχους επανάληψης και δομές ελέγχου. Για παράδειγμα:

```
for (i = 0 to 5) {
  print i * 2;
}

print "Ended loop";
```

Πρόγραμμα 7: Αγκύλες (1)

Με την χρήση των αγκυλών αποτρέπονται λογικά σφάλματα, τα οποία συνήθως είναι και χρονοβόρα να εντοπιστούν, όπως όταν ένα ερωτηματικό τερματισμού εισάγεται κατά λάθος στον πηγαίο κώδικα (Αλγόριθμος 8)

```
 for (i = 0; i < 5; ++i);
    printf("\%d \n", i*2);    /* The incorrect indentation hides the fact
                                 that this line is not part of the loop body. */

 printf("Ended loop");
```

Πρόγραμμα 8: Αγκύλες (2)

Ένα παρόμοιο λάθος είναι όταν προστίθεται μια επιπλέον γραμμή πριν την πρώτη γραμμή (αλγόριθμος 9)

```
 for (i = 0; i < 5; ++i)
    printf(logfile, "loop reached %d\n", i);
    printf("%d\n", i*2);    /* The incorrect indentation hides the fact
                               that this line is not part of the loop body. */
  printf("Ended loop");
```

Πρόγραμμα 9: Αγκύλες (3)

Γι' αυτό το λόγο έχει καθιερωθεί η χρήση των αγκυλών όπως στον αλγόριθμο 10.

```
for( index = 0 ; index < size ; ++index )
{
    arrayA[ index ] = arrayB[ index ] ;
}
```

Πρόγραμμα 10: Αγκύλες (4)

Στον αλγόριθμο 10 φαίνεται σαφώς τόσο η αρχή όσο και το τέλος τους μπλοκ εντολών. Υπάρχουν και άλλες μεθοδολογίες τοποθέτησης των αγκυλών (όπως π.χ. η αρχική αγκύλη να μπαίνει αμέσως μετά την βρόγχο επανάληψης όπως και στον αλγόριθμο 7) οι οποίες δεν διαφέρουν από τον τρόπο που περιγράφτηκε στον παραπάνω αλγόριθμο. Η μικροδιαφορές αυτές δεν επηρεάζουν την αναγνωσιμότητα του κώδικα αλλά θα πρέπει να υπάρχει συνέπεια στην χρήση των αγκυλών [24].

3. Γενικές πρακτικές προγραμματισμού

Κατά την διαδικασία εγγραφής του πηγαίου κώδικα, οι προγραμματιστές έχουν τρία βασικά εργαλεία για να δείξουν τις προθέσεις τους: (i) τα σχόλια (εξήγηση του πηγαίου κώδικα), (ii) σαφή ονόματα μεταβλητών, σταθερών, συναρτήσεων κ.λ.π. (οι λέξεις του ίδιου του προγράμματος), (iii) και το τυπογραφικό στυλ που αναπτύχθηκε στην προηγούμενη ενότητα. Κάθε ένα από αυτά τα στοιχεία βοηθά στην επικοινωνία μεταξύ του του προγραμματιστή που έγραψε τον κώδικα και του προγραμματιστή που διαβάζει τον πηγαίο κώδικα.

Σε αυτή την ενότητα θα παρατεθούν γενικές πρακτικές προγραμματισμού οι οποίες βοηθούν στην καλύτερη κατανόηση του πηγαίου κώδικα.

3.1 Ονοματολογία (αγγλ. Naming conventions)

Η επιλογή των ονομάτων (των αρχείων, των μεταβλητών, των συναρτήσεων, κ.λ.π.) είναι από τα πιο σημαντικά στοιχεία που κάνουν τον πηγαίο κώδικα κατανοητό. Ο τρόπος με τον οποίο γράφουμε ένα όνομα μπορεί να μας πληροφορήσει άμεσα τί αντιπροσωπεύει η οντότητα αυτή όπως αν είναι μεταβλητή, συνάρτηση, σταθερά, κ.λ.π., χωρίς να χρειάζεται να αναζητήσουμε την δήλωση της οντότητας αυτής. Ακόμα τα ονόματα θα πρέπει να έχουν νόημα και να δίνουν στο προγραμματιστή την δυνατότητα να κατανοήσει και να διαβάσει τον πηγαίο κώδικα με ευκολία. Ένα όνομα το οποίο φαίνεται χαριτωμένο ή δακτυλογραφείται εύκολα μπορεί αργότερα να προκαλέσει προβλήματα σε κάποιον που προσπαθεί να κατανοήσει τον κώδικα. Ο πηγαίος κώδικας διαβάζεται πολλές φορές ενώ γράφεται μόνο μία. Γι' αυτό το λόγο η επιλογή της ονοματολογίας παίζει καθοριστικό ρόλο και το όνομα μιας μεταβλητής θα πρέπει να περιγράφει πλήρως και με ακρίβεια την μεταβλητή που αντιπροσωπεύει [18, 14, 11].

Οι κανόνες της ονοματολογίας είναι αρκετά αυθαίρετοι, και γι' αυτό το λόγο οι κανόνες που παραθέτονται πιο κάτω είναι περισσότερο προτροπές ή συμβάσεις που έχουν υιοθετηθεί από ένα μεγάλο κομμάτι προγραμματιστών και όχι τόσο κανόνες. Αυτό που έχει σημασία είναι η συνέπεια τήρησης των οποίων αποφάσεων έχουν παρθεί ή έχουν χρησιμοποιήσει οι αρχικοί δημιουργοί του πηγαίου κώδικα [11].

3.1.1 Γενικοί κανόνες [11, 18, 23, 24, 16]

- Μια αποτελεσματική τεχνική για την εύρεση αντιπροσωπευτικών ονομάτων είναι η δήλωση με λόγια τι αντιπροσωπεύει η μεταβλητή.
- Τα ονόματα που επιλέγονται πρέπει να είναι αντιπροσωπευτικά, έτσι ώστε ο κώδικας να είναι άμεσα κατανοητός. Δεν θα πρέπει να χρησιμοποιούνται συντομογρα-

φίες που είναι ασαφείς ή άγνωστες σε όσους διαβάζουν τον πηγαίο κώδικα για πρώτη φορά. Ακόμα δεν θα πρέπει να γίνονται συντμήσεις διαγράφοντας γράμματα από λέξεις. Πιο κάτω ακολουθούν μερικά "καλά" και "κακά" ονόματα μεταβλητών.

```
int price_count_reader;     // No abbreviation.
int num_errors;             // "num" is a widespread convention.
int num_dns_connections;    // Most people know what "DNS" stands for.

int n;                      // Meaningless.
int nerr;                   // Ambiguous abbreviation.
int n_comp_conns;           // Ambiguous abbreviation.
int wgc_connections;        // Only your group knows what this stands for.
int pc_reader;              // Lots of things can be abbreviated "pc".
int cstmr_id;               // Deletes internal letters.
int temp;                   // Is it temperature? Is it temporary?
```

- Το μήκος μια μεταβλητής θα πρέπει να αντικατοπτρίζει και την εμβέλεια της. Μια τοπική μεταβλητή έχει περιορισμένη εμβέλεια, επομένως και το όνομα δεν χρειάζεται να είναι και πολύ περιγραφικό. Αντιθέτως μία παγκόσμια μεταβλητή, η εμβέλεια της οποίας είναι ευρεία, θα πρέπει να έχει ένα αντιπροσωπευτικό και περιγραφικό όνομα για να ξεχωρίζει.
- Οι συντομογραφίες και τα ακρωνύμια δεν πρέπει να είναι κεφαλαία όταν χρησιμοποιούνται ως ονόματα μεταβλητών.
- Στους επαναληπτικούς βρόχους θα πρέπει να χρησιμοποιούνται ονομασίες με νόημα όπως φαίνεται και στο παρακάτω παράδειγμα:

```
for (window_index = 0; window_index <= window_count; ++window_index)
{
  ...
}
```

3.1.2 Κανόνες ονοματολογίας

Ο πίνακας 3.1 αναφέρει μερικές από τους πιο διαδεδομένους κανόνες που χρησιμοποιούνται στην ονοματολογία των οντοτήτων. Κατά κύριο λόγο περιλαμβάνονται κανόνες που εφαρμόζονται στην C++ αλλά μπορούν να γενικευτούν και στις υπόλοιπες γλώσσες προγραμματισμού.

Οντότητα	Περιγραφή	Παράδειγμα
Αρχεία	Για τα αρχεία διαλέγουμε ένα όνομα το οποίο αντικατοπτρίζει το περιεχόμενο του αρχείου. Τα ονόματα των αρχείων θα πρέπει να είναι όλα πεζά και μπορεί να περιλαμβάνουν την κάτω παύλα (_) ή την άνω παύλα (-). Τα αρχεία προγραμμάτων της c++ θα πρέπει να τελειώνουν σε .cpp και της σε C θα πρέπει να τελειώνουν σε .c ενώ τα αρχεία κεφαλίδων θα πρέπει να τελειώνουν σε .h	`my_useful_class.cpp` `my-useful-class.cpp` `myusefulclass.h` `myusefulclass_test.h`
Defines & Macros	Οι δηλώσεις επεξεργαστή και οι μακροεντολές θα πρέπει να είναι με κεφαλαία γράμματα. Ακόμα οι δηλώσεις επεξεργαστή για τα αρχεία θα πρέπει να τελειώνουν με κάτω παύλα (_).	`#ifndef MODULE_NAME_FILE_NAME_HPP_` `#define MODULE_NAME_FILE_NAME_HPP_` `// the code` `#endif // MODULE_NAME_FILE_NAME_HPP_`
Μεταβλητές	Τα ονόματα των μεταβλητών (αγγλ. variables) είναι όλα πεζά, με παύλες μεταξύ των λέξεων. Για τις παγκόσμιες μεταβλητές δεν υπάρχουν κάποιες ειδικές απαιτήσεις, μιας και η χρήση τους είναι σπάνια, αλλά σε κάθε περίπτωση αν χρησιμοποιείται κάποια, καλό θα ήταν να ξεχωρίζεται προτάσσοντας ένα g_ ή κάποιο άλλο χαρακτηριστικό το οποίο θα τις ξεχωρίζει από τις τοπικές μεταβλητές.	`string table_name; // OK – uses underscore.` `string tableName; // Bad – mixed case.` `int g_a_global_variable;`

Συνέχεια στην επόμενη σελίδα

<table>
<tr><th>Οντότητα</th><th>Περιγραφή</th><th>Παράδειγμα</th></tr>
<tr><td>Συναρτήσεις</td><td>Οι συναρτήσεις (αγγλ. functions) θα πρέπει να είναι camelCased[3]και οι μεταβλητές να είναι όλες πεζές, με κάτω παύλες (_) μεταξύ των λέξεων.

Ο τύπος επιστροφής της κάθε συνάρτησης θα πρέπει να τοποθετείται σε διαφορετική γραμμή

Πρέπει να διαλέγετε ως όνομα ένα ρήμα το οποίο αντανακλά την δράση της συνάρτησης. Καλό είναι να επιλέγονται ονόματα τα οποία αντανακλούν στοιχεία του προβλήματος και όχι την επίλυση του προβλήματος.</td><td><pre>int
applyExample (int example_arg);

void
checkForErrors();</pre></td></tr>
<tr><td>Ονόματα τύπων</td><td>Τα ονόματα των τύπων (αγλλ. type names) είναι CamelCased. Δηλαδή ξεκινούν με κεφαλαίο γράμμα και κάθε νέα λέξη να ξεκινά επίσης με κεφαλαίο, χωρίς να προηγείται κάτω παύλα (_).</td><td><pre>// classes and structs
class UrlTable
{ ...

class UrlTableTester
{ ...

struct UrlTableProperties
{
  string name;
  int num_entries;
}

// typedefs
typedef hash_map<
    UrlTableProperties *,
    string> PropertiesMap;

// enums
enum UrlTableErrors
{ ...</pre></td></tr>
<tr><td colspan="3">Συνέχεια στην επόμενη σελίδα</td></tr>
</table>

[3]CamelCase είναι η πρακτική της γραφής σύνθετων λέξεων ή φράσεων, έτσι ώστε κάθε λέξη ή σύντμηση να αρχίζει με ένα κεφαλαίο γράμμα. Αυτό επιτρέπει την μείωση του μεγέθους των φράσεων μιας και δεν χρησιμοποιούνται κενά ή κάποιοι άλλοι ειδικοί χαρακτήρες μεταξύ των λέξεων (π.χ. "_" ανάμεσα στις λέξεις [29, 33]).

Οντότητα	Περιγραφή	Παράδειγμα
Δομές	Τα μέλη μίας δομής πρέπει να ονομάζονται σαν κανονικές μεταβλητές.	`struct UrlTableProperties { string name; int num_entries; }`
Enum	Τα μέλη των απαριθμητών (αγγλ. enumerations) θα πρέπει να είναι κεφαλαία, με παύλες μεταξύ των λέξεων. Με τα κεφαλαία γράμματα οι σταθερές αυτές ξεχωρίζουν σε ένα πρόγραμμα και δεν μπερδεύονται με τις απλές μεταβλητές.	`enum Color { COLOR_RED, COLOR_GREEN, COLOR_BLUE };`
Κλάσεις (1)	Τα τμήματα μιας κλάσης πρέπει να ταξινομούνται ως: δημόσια, προστατευμένα (αγγλ.public, protected) και ιδιωτικά (αγλλ. private). Έτσι οι προγραμματιστές που θέλουν απλώς να χρησιμοποιήσουν την κλάση να μην χρειάζεται να διαβάζουν και τα ιδιωτικά μέλη της κλάσης. Τα ιδιωτικά μέλη των κλάσεων είναι όπως και οι απλές μεταβλητές, μόνο που τελειώνουν με κάτω παύλα (_). Επισημαίνοντας το εύρος χρησιμοποιώντας κάτω παύλα καθιστά εύκολο τον διαχωρισμό τον μεταβλητών των κλάσεων από τοπικές μεταβλητές που χρησιμοποιούνται.	`class ExampleClass { public: ... protected: ... private: int example_name_; }`

Συνέχεια στην επόμενη σελίδα

Οντότητα	Περιγραφή	Παράδειγμα
Κλάσεις (2)	Μία κλάση πρέπει δηλώνετε σε ένα αρχείο επικεφαλίδας (αγγλ. header file) και να ορίζεται σε ένα πηγαίο αρχείο (αγγλ. source file) όπου τα ονόματα των αρχείων ταιριάζουν με το όνομα της κλάσης καθιστώντας εύκολη την εύρεση των αντίστοιχων αρχείων. Το αρχείο επικεφαλίδας πρέπει να ορίζει μία διεπαφή, και το πηγαίο αρχείο να την υλοποιεί. Τα αρχεία που υλοποιούν τις κλάσεις έχουν κατάληξη .hpp . Οι συναρτήσεις ερωτημάτων (αγγλ. accessors,mutators ή συναρτήσεις get και set) πρέπει να ταιριάζουν με το όνομα της μεταβλητής που προελαύνουν.	``` class ExampleClass { public: ... int getNumEntries() const { return num_entries_; } void setNumEntries (int num_entries) { num_entries_ = num_entries; } protected: ... private: int num_entries_; int example_name_; } ```

Πίνακας 3.1: Ονοματολογια

3.1.3 Υπο-φάκελοι

Στον πίνακα 3.2 φαίνεται η τυπική δομή των φακέλων ενός τυπικού προγράμματος. Ομοίως και εδώ οι οι φάκελοι αυτοί είναι οι πιο χρησιμοποιούμενοι φάκελοι.

Υπο-φάκελος	Περιεχόμενα
include	Τα αρχεία κεφαλίδων.
include/impl/	Τα αρχεία που υλοποιούν τις κλάσεις.
src	Τα πηγαία αρχεία που χρειάζονται για την μεταγλώττιση του προγράμματος.
build	Τα εκτελέσιμα αρχεία του προγράμματος.
doc	Τα αρχεία τεκμηρίωσης του προγράμματος.

Πίνακας 3.2: Υπο-φάκελοι του πηγαίου κώδικα [3, 22]

3.2 Σχολιασμός και τεκμηρίωση (αγγλ. Commenting and Documentation)[27]

Η τεκμηρίωση και ο σχολιασμός του πηγαίου κώδικα είναι μία από τις πιο σημαντικές διαδικασίες κατά την παραγωγή του. Αν και δεν συνδέεται άμεσα με την λειτουργικότητα του κώδικα, συνδέεται έμμεσα με την δυνατότητα επέκτασης του, μιας και παρέχει με διορατικό τρόπο τις προθέσεις του αρχικού προγραμματιστή.

Κατάλληλα θέματα για την τεκμηρίωση συχνά περιλαμβάνουν:

- Γενική περιγραφή του προγράμματος. Πρέπει να περιλαμβάνει:
  - Μία επισκόπηση για το τον σκοπό που δημιουργήθηκε το πρόγραμμα.
  - Ένα απλό παράδειγμα χρήσης του προγράμματος.
  - Οδηγίες εκμάθησης για την βασική χρήση του προγράμματος.
- Τεκμηρίωση του κώδικα. Η τεκμηρίωση του κώδικα περιλαμβάνει:
  - Περιγραφή των κλάσεων και των συναρτήσεων του προγράμματος.
  - Την σχέση μεταξύ των κλάσεων και των συναρτήσεων.
  - Για κάθε συνάρτηση, ανάλογα με την περίπτωση, μία περιγραφή, τις απαιτήσεις για την σωστή λειτουργία της, τα αποτελέσματα της, την τιμή που επιστρέφει καθώς και τυχόν σφάλματα που μπορεί να συμβούν.
  - Εντοπισμό σφαλμάτων και στρατηγικές για την αποφυγή τους.
  - Πως μεταγλωττίζεται και συνδέεται το πρόγραμμα.
  - Η εκδοσή του προγράμματος και τυχόν αλλαγές που έχουν συμβεί.
  - Αιτιολόγηση για τη λήψη αποφάσεων σχεδιασμού.

– Τυχόν ευχαριστίες.

Με τα σχόλια και την τεκμηρίωση διασφαλίζεται ότι ο κώδικας έχει περιγραφτεί με επαρκείς λεπτομέρειες έτσι ώστε όταν κάποιος κοιτάζει τον πηγαίο κώδικα θα μπορεί εύκολα να καταλάβει το σχεδιασμό και το σκοπό του κώδικα [27].

3.2.1 Γενικά σχόλια [25, 24, 28, 11]

- Σε γενικές γραμμές, χρειάζεται τα σχόλια να λένε **ΤΙ** κάνουν στον κώδικα και όχι **ΠΩΣ** το κάνουν.
- Τα καλύτερα σχόλια στον πηγαίο κώδικα είναι ο ίδιος ο πηγαίος κώδικας. Γι' αυτό το λόγο ο κώδικας θα πρέπει να "αυτο-τεκμηριώνεται". Είναι προτιμητέο να εξηγείτε κάτι μέσα από τον ίδιο τον πηγαίο κώδικα (π.χ. με την χρήση κάποιου πολύπλοκου ονόματος μεταβλητής) παρά να χρησιμοποιούνται αργότερα σχόλια τα οποία εξηγούν τί κάνει ο συγκεκριμένος κώδικας.
- Σχόλια πρέπει να τοποθετούνται (i) σε πολύπλοκα, (ii) μη προφανή, (iii) ενδιαφέροντα ή (iv) σημαντικά σημεία του πηγαίου κώδικα. Πριν από αυτά τα σημεία θα πρέπει να τοποθετούνται μικρά σχόλια τα οποία επεξηγούν τί κάνει ο κώδικας. Παραδείγματος χάρη:

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++)
{
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

- Αν τα σχόλια εκτείνονται σε αρκετές γραμμές, μπορεί η στοίχιση τους να τα κάνει πιο ευανάγνωστα. Παραδείγματος χάρη:

```
DoSomething();                    // Comment here so the comments line up.
DoSomethingElseThatIsLonger();    // Comment here so there are two spaces
                                  // between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
DoSomething(); /* For trailing block comments, one space is fine. */
```

3.2.2 Σχόλια οντοτήτων

Ο πίνακας 3.3 αναφέρει μερικούς από τους πιο διαδεδομένους τρόπους που χρησιμοποιούνται για να σχολιάσουν τις οντότητες του πηγαίου κώδικα.

<table>
<tr><th>Οντότητα</th><th>Περιγραφή</th><th>Παράδειγμα</th></tr>
<tr><td>Αρχεία</td><td>Κάθε αρχείο πρέπει να περιέχει την άδεια χρήσης του πηγαίου κώδικα (π.χ. Apache 2.0, BSD, LGPL, GPL). Η επιλογή της κατάλληλης άδειας χρήσης είναι πολύπλοκη διαδικασία η οποία όμως δεν πρέπει να αμελείται.
Πρέπει να περιέχεται ο συγγραφέας του πηγαίου.
Κάθε αρχείο πρέπει να έχει ένα σχόλιο το οποίο περιγράφει τα περιεχόμενα του.</td><td><pre>/*
Copyright (c) 2013,
    Anagnostopoulos Vasilis—
    Thanos, Post—Graduate
    Student in University of
    Piraeus. All rights
    reserved.

Redistribution and use in
    source and binary forms,
    with or without
    modification,
are permitted provided that the
     following conditions are
    met:
  ...
*/

/*
This is a program that ...
*/</pre></td></tr>
<tr><td>Μεταβλητές</td><td>Σε γενικές γραμμές το όνομα μίας μεταβλητής θα πρέπει να είναι αρκετά περιγραφικό ώστε να δίνει την δυνατότητα στον προγραμματιστή να καταλάβει για ποιο λόγο χρησιμοποιείται η συγκεκριμένη μεταβλητή. Σε ορισμένες περιπτώσεις, μερικά παραπάνω σχόλια χρειάζονται.</td><td><pre>// The total number of tests
    cases that we run through
    in this regression test.
const int k_num_test_cases = 6;</pre></td></tr>
<tr><td colspan="3">Συνέχεια στην επόμενη σελίδα</td></tr>
</table>

Οντότητα	Περιγραφή	Παράδειγμα
Συναρτήσεις (1)	Κάθε δήλωση συνάρτησης θα πρέπει να έχει σχόλια τα οποία προηγούνται της συνάρτησης και περιγράφουν τί κάνει η συνάρτηση και πως να χρησιμοποιείται. Γενικά τα σχόλια στην δήλωση μίας συνάρτησης περιγράφουν την χρήση της συνάρτησης, ενώ τα σχόλια στην υλοποίηση μίας συνάρτησης περιγράφουν την λειτουργίας της. Κατά την δήλωση μίας συνάρτησης πρέπει να αναφέρονται: • Οι μεταβλητές εισόδου και οι μεταβλητές που επιστρέφει η συνάρτηση. • Εάν η συνάρτηση δεσμεύει μνήμη την οποία ο προγραμματιστής θα πρέπει να την αποδεσμεύσει (αυτό ισχύει σε γλώσσες που δεν έχουν συλλογή απορριμάτων (αγγλ. garbage collection) όπως η C++). • Εάν κάποια από τις μεταβλητές εισόδου ή εξόδου θα πρέπει να είναι δείκτης στο κενό. • Εάν υπάρχει οποιαδήποτε επίπτωση στην απόδοση με την χρησιμοποίηση της συνάρτησης.	``` // Returns an iterator for this table. It is the client's // responsibility to delete the iterator when it is done with it, // and it must not use the iterator once the GargantuanTable object // on which the iterator was created has been deleted. // // The iterator is initially positioned at the beginning of the table. // // This method is equivalent to: // Iterator* iter = table->NewIterator(); // iter->Seek(""); // return iter; // If you are going to immediately seek to another place in the // returned iterator, it will be faster to use NewIterator() // and avoid the extra seek. Iterator* GetIterator() const; ```

Συνέχεια στην επόμενη σελίδα

Οντότητα	Περιγραφή	Παράδειγμα
Συναρτήσεις (2)	Εάν υπάρχει κάποιο πολύπλοκο κομμάτι για το πως μίας συνάρτηση κάνει την δουλειά της τότε θα πρέπει να υπάρχει ένα επεξηγηματικό σχόλιο στην δήλωση της συνάρτησης το οποίο θα περιγράφει (i) οποιοδήποτε κόλπα χρησιμοποιήθηκαν κατά την δημιουργία της συνάρτησης αυτής, (ii) θα παρουσιάζει συνοπτικά τα βήματα που εκτελούνται κατά την λειτουργία της συνάρτησης (iii) και θα εξηγεί γιατί δεν προτιμήθηκε κάποιος άλλος τρόπος υλοποίησης. . Όταν σε μία συνάρτηση περνιούνται σαν ορίσματα μεταβλητές boolean, ή ακέραιες μεταβλητές, ή κενοί δείκτες θα πρέπει να υπάρχουν σχόλια τα οποία θα επεξηγούν τί κάνουν αυτές οι τιμές.	`bool success = CalculateSomething( interesting_value, 10, // Default base value. false, // Not the first time we're calling this. NULL); // No callback.`

Συνέχεια στην επόμενη σελίδα

Οντότητα	Περιγραφή	Παράδειγμα
Κλάσεις	Κάθε κλάση πρέπει να έχει ένα συνοδευτικό σχόλιο το οποίο θα περιγράφει την κλάση και πως πρέπει να χρησιμοποιείται. Τα μέλη των συναρτήσεων θα πρέπει να έχουν ένα σχόλιο το οποίο περιγράφει την χρήση τους. Ακόμα αν μπορούν να λάβουν τιμές με ιδιαίτερη σημασία θα πρέπει να αναφέρονται στα σχόλια. Γενικά ένα αρχείο επικεφαλίδας θα περιγράφει τις κλάσεις που δηλώνονται μέσα στο αρχείο με μία επισκόπηση στο τί κάνει η κάθε κλάση και πως χρησιμοποιείται. Αντιθέτως ένα πηγαίο αρχείο πρέπει να περιέχει περισσότερες πληροφορίες για την υλοποίηση της κλάσης ή πολύπλοκων αλγορίθμων.	`// Iterates over the contents of a GargantuanTable. Sample usage:` `//  GargantuanTableIterator* iter = table->NewIterator();` `//  for (iter->Seek("foo"); !iter->done(); iter->Next()) {` `//  process(iter->key(), iter->value());` `//  }` `//  delete iter;` `class GargantuanTableIterator {` `...` `private:` `// Keeps track of the total number of entries in the table.` `// Used to ensure we do not go over the limit. -1 means` `// that we don't yet know how many entries the table has.` `int num_total_entries_;` `};`

Πίνακας 3.3: Σχόλια

3.3 Συνέπεια

Μία από τις πτυχές για την ποιοτική συγγραφή κώδικα είναι η συνέπεια (αγγλ. Consistency) στις αποφάσεις που έχουν ληφθεί , που συνεπάγεται ότι η η ποιότητα είναι αντιστρόφως ανάλογης των παρακλήσεων. Ο κώδικας που έχει πολλές περιττές παραλλαγές, είναι ο κώδικας στον οποίο χάνετε πολύ χρόνος εργασίας. Η συνέπεια είναι μια γενική αξία, και ως εκ τούτου, ακόμη και αν μία επιλογή θα μπορούσε να είναι καλύτερη σε μία συγκεκριμένη περίπτωση, το κόστος όμως για την συνέπεια πολλές φορές αντισταθμίζει τα τυχόν πλεονεκτήματα που ίσως υπάρχουν. Ωστόσο, μερικές φορές γίνο-

νται εξαιρέσεις αν υπάρχουν ιδιαίτερα ελκυστικοί λόγοι ή σε προβλεπόμενες συνθήκες που έχουν θεσπιστεί κάποιες άλλες κατευθυντήριες γραμμές [31].

Συνέπεια στην συγγραφή του πηγαίου κώδικα είναι η χρησιμοποίηση των ίδιων προτύπων στις ίδιες καταστάσεις. Παραδείγματος χάρη, στην ονοματολογία η χρησιμοποίηση του ίδιου ονόματος, όταν παρουσιάζεται ο ίδιος τύπος κατάστασης. Αν χρησιμοποιούνται οι μεταβλητές "i","index","inx" κ.λ.π. στους βρόγχους επανάληψης τότε δεν έχουμε συνέπεια. Αντιθέτως, αν χρησιμοποιείται μόνο η μεταβλητή "index" τότε παράγεται κώδικας που είναι πολύ πιο απλώς και γρήγορος στην κατανόηση χωρίς περιττό "θόρυβο".[27]

3.4 Μαγικοί αριθμοί [27, 23, 16]

Η πρακτική της ενσωμάτωσης δεδομένων εισόδου ή οποιοδήποτε άλλων δεδομένων στον πηγαίο κώδικα συναντάται στην βιβλιογραφία ως "μαγικοί αριθμοί" (αγγλ. magic numbers ή hard coding) και αποτελεί μία κακή προγραμματιστική τεχνική. Η χρησιμοποίηση τέτοιων αριθμών παράγουν κώδικα ο οποίος είναι δύσκολο να συντηρηθεί καθώς δεν δίνουν κάποια χρήσιμη πληροφορία σε κάποιον που διαβάζει τον πηγαίο κώδικα και είναι δύσκολο να αλλάξουν με συστηματικό τρόπο, αν είναι διεσπαρμένες μέσα στον πηγαίο κώδικα.

Εάν οι αριθμοί που χρησιμοποιούνται στον πηγαίο κώδικα δεν έχουν μία προφανή σημασία από μόνοι τους, τότε η αναγνωσιμότητα του πηγαίου κώδικα μπορεί να βελτιωθεί χρησιμοποιώντας μεταβλητές στην θέση των αριθμών. Εναλλακτικά μπορεί να χρησιμοποιηθεί και μία συνάρτηση που επιστρέφει την επιθυμητή τιμή. Στο πρόγραμμα 25 φαίνεται ένα ακριβώς τέτοιο παράδειγμα στο οποίο έχουν ενσωματωθεί "μαγικοί αριθμοί" μέσα στον πηγαίο κώδικα.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
int
main()
{
  int fahr;

  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
    printf("%3d␣%6.1f\n", fahr, (5.0/9.0)*(fahr -32));

  return 0;
}
```

Πρόγραμμα 25: Παράδειγμα "Μαγικών αριθμών" (1)

Αντιθέτως στο πρόγραμμα 26 που πλέον οι αριθμοί μέσα στο πηγαίο κώδικα έχουν αντι-

κατασταθεί από μεταβλητές φαίνεται ξεκάθαρα η χρησιμότητα τους.

```c
#include <stdio.h>
#define LOWER 0 /* lower limit of table */
#define UPPER 300 /* upper limit */
#define STEP 20 /* step size */

/* print Fahrenheit-Celsius table */
int
main()
{
  int fahr;

  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
    printf("%3d␣%6.1f\n", fahr, (5.0/9.0)*(fahr -32));

  return 0;
}
```

Πρόγραμμα 26: Παράδειγμα "Μαγικών αριθμών" (2)

3.5 Σύγκριση με βάση το αριστερό μέλος [27, 6, 13]

Στις γλώσσες προγραμματισμού που χρησιμοποιούν διαφορετικά σύμβολα για την ανάθεση τιμής (συνήθως ένα σύμβολο ίσον (=)) και για την σύγκριση (συνήθως δύο ίσον (==)) (π.χ. C/C++, Java και η πλειοψηφία των γλωσσών τα τελευταία 15 χρόνια), και όπου οι αναθέσεις τιμών είναι συντακτικά σωστές μέσα σε δομές ελέγχου ή επαναληπτικούς βρόχους υπάρχει ένα πλεονέκτημα στην υιοθέτηση της σύγκρισης με βάσης το αριστερό μέρος και την τοποθέτηση των σταθερών ή των εκφράσεων προς τα αριστερά σε κάθε σύγκριση.

Στα προγράμματα 27 και 28 φαίνονται τα πλεονεκτήματα της υιοθέτησης της σύγκρισης με βάση το αριστερό μέλος. Και στις δύο περιπτώσεις θέλουμε να συγκρίνουμε την τιμή του a με το 42. Στο πρώτο πρόγραμμα δεν υπάρχει κάποια διαφορά μιας και στις δύο περιπτώσεις η σύγκριση πραγματοποιείται σωστά.

```c
if (a == 42) { ... }  // A right-hand comparison checking if a equals 42.
if (42 == a) { ... }  // Recast, using the left-hand comparison style.
```

Πρόγραμμα 27: Παράδειγμα σύγκρισης με βάση το αριστερό μέλος (1)

Η διαφορά και το πλεονέκτημα της σύγκρισης με βάση το αριστερό μέρος προκύπτει όταν ένας προγραμματιστής πληκτρολογήσει = αντί για ==.

```c
if (a = 42) { ... }  // Inadvertent assignment which is often hard to debug
if (42 = a) { ... }  // Compile time error indicates source of error
```

Πρόγραμμα 28: Παράδειγμα σύγκρισης με βάση το αριστερό μέλος (2)

Στην πρώτη περίπτωση λοιπόν η τιμή του a θα αλλάξει σε 42, με αποτέλεσμα ο κώδικας μέσα στην δομή ελέγχου if να εκτελείται πάντα και καθώς η έκφραση αυτή είναι συντακτικά σωστή, το σφάλμα μπορεί να περάσει απαρατήρητο από το προγραμματιστή, και το λογισμικό μπορεί να κυκλοφορήσει με σφάλμα. Αντιθέτως στην δεύτερη περίπτωση θα έχουμε σφάλμα κατά την μεταγλώττιση του προγράμματος, καθώς δεν μπορούμε να αναθέσουμε τιμές σε αριθμητικές τιμές. Επομένως το σφάλμα θα διορθωθεί σίγουρα.

4. Εφαρμογή και συμπεράσματα

Αντί επιλόγου προτιμήθηκε να γραφτεί ένα πρότυπο προγράμματος το οποίο συνοψίζει όλες τις παραπάνω οδηγίες. Αυτό που θα ήθελα να τονίσω κλείνοντας είναι πως όλα τα παραπάνω δεν είναι αναγκαστικοί κανόνες αλλά περισσότερο προτροπές ή συμβάσεις που έχουν υιοθετηθεί από ένα μεγάλο κομμάτι προγραμματιστών. Δεν έχει τόσο σημασία αν κάποιος θα αφήσει δύο διαστήματα ή την αγκύλη θα την βάζει σε ξεχωριστή γραμμή όσο το να είναι συνεπής σε αυτές τις αποφάσεις.

Το νόημα της ύπαρξης κάποιων κατευθυντήριων γραμμών για την σύνταξη προγραμμάτων είναι για να μπορούν οι προγραμματιστές να έχουν ένα κοινό "λεξιλόγιο", ώστε να μπορούν να επικεντρωθούν σε αυτό που λέει το πρόγραμμα και όχι στο πως το λέει.

Παρακάτω ακολουθεί ένα πρόγραμμα το οποίο τηρεί το σύνολο των παραπάνω κανόνων.

```
/*
Copyright (c) 2013, Anagnostopoulos Vasilis-Thanos, Post-Graduate Student in
    University of Piraeus. All rights reserved.

Redistribution and use in source and binary forms, with or without
    modification,
are permitted provided that the following conditions are met:
  ...
*/

#ifndef TEST_H_
#define TEST_H_

#define SOME_MACRO_                                 // all uppercase ending with
    trailing underscore

typedef double SomeType;                           // CamelCase, starting capital

/*
This is a class that ...
*/
class SomeClass
{                                 // camelcase, starting capital
  public:
    void method();
    Real anotherMethod(Real x,                     // camelcase, starting lowercase
                       Real y) const;
    void setMember(Real);                          // setter, leading "set"

  private:
    //Some comment describing the variables
    Real member_;                                  // all lowercase with underscore
```

```
        between the words
    Integer another_member_;                    // trailing underscore
};

struct SomeStruct
{
  Real foo_foo;                                 // struct members:
  Integer bar;                                  // no trailing underscore
};

/*
This is a function that ...
*/
Size
someFunction(Real parameter,                  // one parameter per line,
             Real another_parameter)           // camelCase, starting lowercase
{
  Real local_variable = 0.0;
  if (condition)
  {                                             // brackets here...
      local_variable += 3.14159;
  }
  else
  {                                             // ...here...
      local_variable -= 2.71828;
  }                                             // ...and here.
  return 42;
}

#endif // TEST_H_
```

Αναφορές

[1] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

[2] A. Ayerbe and I. Vazquez. Software products quality improvement with a programming style guide and a measurement process. In *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*, pages 172–178, 1998.

[3] Boost. Boost library requirements and guidelines. http://www.boost.org/development/requirements.html. [Πρόσβαση στις 21 Δεκμβρίου 2013].

[4] Thomas C. Bressoud. C++ programming style guide. http://coding.

smashingmagazine.com/2012/10/25/why-coding-style-matters/. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[5] A. Cox and M. Fisher. Programming style: Influences, factors, and elements. In *Advances in Computer-Human Interactions, 2009. ACHI '09. Second International Conferences on*, pages 82–89, 2009.

[6] P.J. Deitel and H.M. Deitel. *C++: How to Program*. Deitel series. Prentice Hall PTR, 2011.

[7] Richard Stallman et all. *GNU Coding Standards*, December 2013.

[8] GCC. Gcc coding conventions. http://gcc.gnu.org/codingconventions.html. [Πρόσβαση στις 21 Δεκμβρίου 2013].

[9] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Agile Software Development. Addison-Wesley, 2003.

[10] GNU. Gnu coding standards. http://www.gnu.org/prep/standards/standards.html. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[11] Google. Google c++ style guide. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[12] Google. Google style guide. http://code.google.com/p/google-styleguide/. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[13] Programming Research Group et al. High-integrity c++ coding standard manual: Version 3.3. http://www.programmingresearch.com/content/misc/hicpp-manual-version-3-3.pdf, 2013. [Πρόσβαση στις 21 Δεκμβρίου 2013].

[14] Taligent Inc. Taligent documentation. http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_3.html. [Πρόσβαση στις 28 Δεκμβρίου 2013].

[15] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1982.

[16] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

[17] Hidetaka Kondoh and Kokichi Futatsugi. To use or not to use the goto statement: Programming styles viewed from hoare logic. *Science of Computer Programming*, 60(1):82 – 116, 2006.

[18] David H. Leserman. Fx-alpha c and c++ coding conventions. http://fxa.noaa.gov/manuals/codingGuidelines.html. [Πρόσβαση στις 24 Δεκμβρίου 2013].

[19] A. Mohan and N. Gold. Programming style changes in evolving source code. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 236–240, 2004.

[20] P. W. Oman and C. R. Cook. A paradigm for programming style research. *SIGPLAN Not.*, 23(12):69–78, December 1988.

[21] Paul W. Oman and Curtis R. Cook. A programming style taxonomy. *Journal of Systems and Software*, 15(3):287 – 301, 1991.

[22] PCL. Pcl c++ programming style guide. http://pointclouds.org/documentation/advanced/pcl_style_guide.php#naming. [Πρόσβαση στις 25 Ιανουαρίου 2014].

[23] Geotechnical Software Services. C++ programming style guidelines. http://geosoft.no/development/cppstyle.html. [Πρόσβαση στις 29 Δεκμβρίου 2013].

[24] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

[25] Linus Torvalds. Linux kernel coding style. https://www.kernel.org/doc/Documentation/CodingStyle. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[26] Yanqing Wang, Bo Zheng, and Hujie Huang. Complying with coding standards or retaining programming style: A quality outlook at source code level. *JSEA*, 1(1):88–91, 2008.

[27] Wikibooks. Computer programming/coding style. http://en.wikibooks.org/wiki/Computer_Programming/Coding_Style. [Πρόσβαση στις 22 Δεκμβρίου 2013].

[28] Wikipedia. Best coding practices — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Best_coding_practices. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[29] Wikipedia. Camelcase — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/CamelCase. [Πρόσβαση στις 30 Δεκμβρίου 2013].

[30] Wikipedia. Coding conventions — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Coding_conventions. [Πρόσβαση στις 21 Δεκμβρίου 2013].

[31] Wikipedia. Programming style — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Programming_style. [Πρόσβαση στις 18 Δεκμβρίου 2013].

[32] Wikipedia. Software quality — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Software_quality_model#Measurement. [Πρόσβαση στις 21 Δεκμβρίου 2013].

[33] YoLinux.com. C++ coding practices, style, standards and document generation (doxygen). http://www.yolinux.com/TUTORIALS/LinuxTutorialC++CodingStyle.html. [Πρόσβαση στις 21 Δεκμβρίου 2013].