PANDORA CODING GUIDELINES

v. 2013-2014

Περιεχόμενα:

1. Εισαγωγή

Η C++ είναι γλώσσα με πολλά ισχυρά γνωρίσματα, αλλά αυτή η ισχύς φέρνει επιπλέον περιπλοκότητα, που με τη σειρά της μπορεί να κάνει τον κώδικα πιο επιρρεπή σε bugs και δυσκολότερο να αναγνωστεί και να συντηρηθεί.

Σε αυτό το κείμενο συγκεντρώνονται οι συμβάσεις που οφείλουν να ακολουθούν τα μέλη της ομάδας κατά τη συγγραφή κώδικα σε C++. Σκοπός των συμβάσεων είναι σε πρώτη φάση η καλή αναγνωσιμότητα του κώδικα καθώς και η ευκολία κατανόησής του από κάποιον που δεν πήρε μέρος στη συγγραφή του.

Ακόμη αναφέρονται κάποιες πρακτικές που χρησιμοποιούν συγκεκριμένα γνωρίσματα της γλώσσας σε συγκεκριμένες μόνο περιπτώσεις, με σκοπό κυρίως την αυξημένη ασφάλεια έναντι σε bugs.

Σημείωση: Δεν πρόκειτα για tutorial στην C++. Θεωρείται ότι είστε εξοικειωμένοι με τη γλώσσα. Για οδηγούς και βιβλία στη γλώσσα βλ. στις αναφορές.

2. Συμβάσεις στην ονοματολογία και μορφολογία

1. Στις περιπτώσεις **κλάσεων** και **μεθόδων**, η ονοματολογία θα πρέπει να έχει άμεση σχέση με τη λειτουργία που επιτελείται ενώ για την περίπτωση μεταβλητών, enums, macros, structs κλπ η ονοματολογία πρέπει να έχει άμεση σχέση με τη χρήση τους.
2. Τα **ονόματα κλάσεων**, **δομών**, **enums** και **template arguments** πρέπει να έχουν **κεφαλαίο** το πρώτο τους γράμμα, **πεζά** τα υπόλοιπα και να χρησιμοποιείται **κεφαλαίο** γράμμα για τον χωρισμό των λέξεων μέσα στο ίδιο όνομα με τα **υπόλοιπα** γράμματα πεζά (**mixed case**).
3. Τα ονόματα **local** και **member μεταβλητών** πρέπει να ξεκινούν με **πεζό** το πρώτο γράμμα. Σε περιπτώσεις που υπάρχουν 2 λέξεις, ο διαχωρισμός γίνεται και εδώ με **mixed case.**
4. Οι **member μεταβλητές** μια κλάσης (**private** και **protected**) τελειώνουν με **κάτω παύλα**.
5. **#define** και **macro** names γράφονται με **κεφαλαία** και ο διαχωρισμός των λέξεων γίνεται και εδώ με χρήση **underscore**.
6. Τα ονόματα που αντιπροσωπεύουν **namespaces** πρέπει να είναι όλα με **πεζά** γράμματα και ο διαχωρισμός των λέξεων γίνεται με χρήση **underscore**.
7. Τα ονόματα **μεθόδων** πρέπει να είναι **ρήματα** για να δηλώνουν την λειτουργία που επιτελούν και να ξεκινούν με **πεζό** το πρώτο γράμμα. Σε περιπτώσεις που υπάρχουν 2 λέξεις, χρησιμοποιείται **mixed case**.
8. Οι μεταβλητές γένους πρέπει να είναι όμοιες με τους τύπους τους. Παραδείγματα:

```
void setTopic(Topic* topic)
        // NOT: void setTopic(Topic* value)
        // NOT: void setTopic(Topic* aTopic)
        // NOT: void setTopic(Topic* t)
void connect(Database* database)
        // NOT: void connect(Database* db)
        // NOT: void connect (Database* oracleDB)
```

9. Η χρήση του "**is**" χρησιμοποιείται στις μεθόδους και στις **Boolean** μεταβλητές για να δείξουν ενέργεια ή/και ιδιότητα.
10. Οι κλάσεις τύπου **Exception** πρέπει να λήγουν σε Exception (π.χ.IOException)
11. Για τις μεταβλητές που χρησιμοποιούνται ως **pointers** ή **references** πρέπει να τοποθετείται το ***** ή **&** δίπλα στον τύπο της μεταβλητής που δείχνει και όχι δίπλα στο όνομα.
12. **Αποφεύγονται** τα μονά γράμματα ώς μεταβλητές.
13. Για **identation** χρησιμοποιούνται αυστηρά παντού **2 κενά**. Όχι tab!
14. Το συνολικό **μήκος** κάθε **σειράς** δεν πρέπει να ξεπερνάει τους **80 χαρακτήρες**.

3. Η μορφή του κώδικα (σχεδιαγράμματα και γραφή)

1. Σε ένα βρόχο ή σε μια συνθήκη γενικά η βασική οδόντωση του κώδικα καλό θα ήταν να περιλαμβάνει 2 γραμμές για να είναι πιο όμορφα δομημένο
2. Η **σειρά** δήλωσης των **members** μιας **κλάσης** έχει συγκεκριμένη μορφή. Προηγούνται οι **public** , **protected**, **private** μέθοδοι και ακολουθούν τα **protected** και **private** πεδία - με αυτή τη σειρά.

```
class MyClass : public OtherClass {
  public:
    MyClass();
    explicit MyClass(int var);
    ~MyClass();
    void someFunction();
    void someFunctionThatDoesNothing();


    void setSomeVar(int var);
    int someVar() const;

  private:
    bool someInternalFunction();

    int someVar_;
    int someOtherVar_;
};
```

3. Καλή για την κατανόηση και την ανάγνωση του κώδικα είναι η χρήση κενών (white spaces) κατά τη γραφή, παρακάτω δίνονται ορισμένα παραδείγματα:

   Στον τρόπο γραφής παραστάσεων:

   ```
   a = ( b + c ) * d; // NOT: a=(b+c)*d
   for ( ii = 0; ii < 10; ii++) { // NOT: for(ii=0;ii<10;ii++){
   ```

   Στον τρόπο γραφής των ορισμάτων των μεθόδων:

   ```
   Matrix4x4 matrix = new Matrix4x4();
   double cosAngle = Math.cos(angle);
   double sinAngle = Math.sin(angle);
   matrix.setElement(1, 1, cos_angle);
   matrix.setElement(1, 2, sin_angle)
   matrix.setElement(2, 1, sin_angle);
   matrix.setElement(2, 2, cos_angle);
   ```

   Γενικά πρέπει να δίνεται ιδιαίτερη προσοχή στην **ευθυγράμμιση** και τακτοποίηση του κώδικα και όπου χρειάζεται να μένουν **κενά** και **κενές γραμμές** για την καλύτερη ανάγνωση του.

4. Σχόλια στον κώδικα - documentation

Τα σχόλια σε έναν κώδικα δεν πρέπει να έχουν μεγάλη έκταση και θα πρέπει να γράφονται στην **αγγλική** γλώσσα.

Πιο συγκεκριμένα :

1. **Κάθε δήλωση** (κλάσης, μεθόδου κλπ) θα πρέπει να συνοδεύεται και από ένα σχόλιο όπου να ορίζεται η συγκεκριμένη λειτουργία του με ακρίβεια
2. Όπως αναφέρθηκε, τα γενικά σχόλια που αφορούν στην επεξήγηση κλάσεων πρέπει να παρατίθενται **πάνω από κάθε κλάση, στην αρχή κάθε header file (description).** Η επεξήγηση της λειτουργίας μεθόδων και συναρτήσεων, παρατίθεται πάνω από κάθε μέθοδο και συνάρτηση στο header file.
3. Οι συμβάσεις που ακολουθούνται για το documentation είναι αυτές που ορίζονται στο Doxygen (http://www.stack.nl/~dimitri/doxygen/). Τα πρότυπα που θα ακολουθήσουμε είναι τα εξής:

- Πριν από κάθε class, struct, enum υπάρχει ένα σχόλιο με την λειτουργικότητά της:

```
/**
@class CrsmSlam
@brief The main slam class. Contains the main functionalities of CRSM slam.
**/

/**
@struct CrsmPoint
@brief Holds the variables for a laser ray casted at a specific point
**/

/**
@enum CrsmDirection
@brief Types of map expansion
**/
```

- Πριν από κάθε συνάρτηση μέλος μίας class/struct υπάρχει documentation της παρακάτω μορφής. Το ίδιο block σχολίων μπαίνει στο .h και στο .cpp.

```
/**
@brief Returns the map occupancy probability of coordinates (x,y)
@details The probability ranges from 0-255 (0 is occupied, 255 is free)
```

```
@param x [int] : The x coordinate
@param y [int] : The y coordinate
@return char probability
@see CrsmMap::p (if needed)
@bug Crashes when… (if needed)
@warning If x,y out of range program… (if needed)
**/
char getMapProbability(int x, int y);
```

- Για κάθε μεταβλητή μέλος μίας class/struct, γίνεται μία σύντομη περιγραφή με τον εξής τροπο:

```
std::vector<CrsmPose> robotTrajectory;        //!< Container for the robot trajectory
std::set<int> bigChanges;   //!< Holds the irregularities of a specific scan in terms of distance
```

Μπορείτε να δείτε τον κώδικα του CRSM SLAM εδώ και το αποτέλεσμα που προκύπτει από το doxygen είναι εδώ.

5. Καλές πρακτικές κώδικα

Οι παρακάτω συμβάσεις, όπως αναφέρθηκε και στην εισαγωγή, δεν περιορίζονται στο να βοηθούν στην αναγνωσιμότητα του κώδικα, αλλά είναι ζωτικές για την προστασία του από bugs. Προσφέρουν μέσω του στατικού ελέγχου από τον compiler μια επιπλέον δικλίδα ασφαλείας - κυρίως από τον εαυτό μας κατά τη συγγραφή του, αλλά και απέναντι σε λάθος χρήση του κώδικά μας από 3ους.

5.1 Δομή κλάσεων

1. Πάντα ορίζονται **constructor** και **destructor**, ακόμα και να είναι κενοί.
2. Στην περίπτωση που ο **constructor** παίρνει **ένα μόνο όρισμα**, χρησιμοποιείται το keyword **explicit** πρίν την δήλωση του.
3. Αν η κλάση έχει **virtual μεθόδους** ο **destructor** πρέπει **πάντα** να δηλώνεται **virtual**.
4. Όταν ξανα-προσδιορίζεται μια κληρονομήσιμη **virtual** μέθοδος, να χρησιμοποιείται ξάνα η λέξη virtual (δεν απαιτείται από τον compiler, ωστόσο βοηθάει, όπως όλα σε αυτό το κείμενο, στην αναγνωσιμότητα).
5. **Καμία** κλάση δεν περιέχει public μεταβλητές. Για άμεση πρόσβαση σε ένα **private** πεδίο χρησιμοποιούνται συναρτήσεις τύπου **get/set**.
6. **Δεν** χρησιμοποιούνται **ποτέ** global μεταβλητές.
7. Ομοίως **δεν** χρησιμοποιούνται global συναρτήσεις. Αν χρειάζεται να οριστεί λειτουργικότητα που δεν σχετίζεται με το state κάποιου αντικειμένου - οπότε και το instantiation ενός κάθε φορά που χρειάζεται δεν έχει νόημα - περικλείεται σε **static** μεθόδους μιας κλάσης.
8. Οι δομές (**struct**) χρησιμοποιούνται μόνο για να οριστούν παθητικά αντικείμενα που θα χρησιμεύουν ώς **δοχεία δεδομένων**. Για όλες τις άλλες περιπτώσεις δημιουργούνται κλάσεις. Αν έχετε αμφιβολία, χρησιμοποιείστε κλάση.
9. Η χρήση των malloc, realloc και free πρέπει να αποφεύγεται και στη θέση τους να χρησιμοποιούνται οι λειτουργίες των **new** και **delete** για τη δημιουργία και καταστροφή δυναμικών αντικειμένων.
10. Το **overloading** των **operators** θα πρέπει να **αποφεύγεται**. Σε γενικές γραμμές μην το επιχειρείτε, πλήν εξαιρετικών περιπτώσεων και εφόσον ξέρετε πολύ καλά τι κάνετε. Καλύτερη εναλλακτική είναι μια μέθοδος που να περιγράφει την αντίστοιχη λειτουργία, π.χ. multiply ή add.
11. Στο declaration των μεθόδων προηγούνται τα ορίσματα εισόδου (αυτά που δίνονται μόνο για ανάγνωση) και έπονται τα ορίσματα εξόδου (για εγγραφή).
12. Ο κώδικας των κλάσεων περιλαμβάνεται πάντα μέσα σε **namespaces**, ώστε να αποφεύγονται name collisions καθώς και για να είναι φανερό από που προέρχονται σε περίπτωση που χρησιμοποιούνται από άλλα πακέτα.
13. **Δε χρησιμοποιείται ποτέ** το keyword **using** , π.χ. using namespace std.
14. Αν το πλήρες όνομα κάποιου τύπου είναι πολύ μεγάλο και **δύσχρηστο** για να χρησιμοποιηθεί (κάτι που μπορεί να συμβεί και όταν έχουμε μεγάλο namespace ή/και

templates) **ορίζουμε typedef** στο εσωτερικό της **δήλωσης** της κλάσης. π.χ. **typedef** actionlib::SimpleActionClient<slam_communications::slamMapAction> MapClient;

5.2 Passing by value, reference or address & const correctness

1. Το πέρασμα ενός αντικειμένου σε μια συνάρτηση by **value** καλό είναι να **αποφεύγεται**, τόσο για λόγους απόδοσης, όσο και για να προληφθούν bugs και side effects (π.χ. στον copy constructor).
2. Όταν θέλουμε να περάσουμε μια μεταβλητή ως όρισμα σε μια μέθοδο για ανάγνωση **(input argument**) χρησιμοποιούμε **const reference (const&)**.
3. Όταν θέλουμε να περάσουμε μια μεταβλητή ως όρισμα σε μια μέθοδο για εγγραφή (**output argument**), χρησιμιοποούμε **pointer (*)**.
4. Επίσης μια μέθοδος μπορεί να **επιστρέφει** χωρίς να υπάρχει πρόβλημα **const&** ή **const*** σε κάποιο **πεδίο** του αντικειμένου (θα πρέπει να προσεχθεί βέβαια να χρησιμοποιηθεί μόνο όσο το αντικείμενο είναι ακόμα ζωντανό). Αντίθετα η επιστροφή ενός πεδίου με μή const & ή * δεν είναι καθόλου καλή ιδέα και πρέπει να αποφεύγεται, καθώς η μεταβολή των πεδίων ενός αντικειμένου είναι καλά να γίνεται μόνο από το ίδιο.
5. Με βάση τα 2 και 3, **όλα** τα ορισματα που περνάνε με **reference** πρέπει να είναι **const**. Για να μπορούν να χρησιμοποιηθούν, πρέπει οι κλάσεις των αντικειμένων που χρησιμοποιούμε να είναι const correct.
6. **const correctness**: Οι **μέθοδοι** της κλάσης που **δεν μεταβάλουν** κάποιο από τα πεδία της θα **ορίζονται const** ωστε να μπορούν να χρησιμοποιηθούν από **const αντικείμενα** . Βλ. εδώ και εδώ για περισσότερα.
7. Για **dynamically allocated** αντικείμενα (με τη χρήση new) είναι καλύτερα να χρησιμοποιούνται **smart_pointers**. Ιδιαίτερα όταν πρέπει να παραδοθεί ή να μοιραστεί η ιδιοκτησία ενός αντικειμένου επιβάλλεται η χρήση **boost::shared_pointer**.
8. Οι κλάσεις που έχουν member variables που είναι pointers θα πρέπει να χρησιμοποιούν **boost::scoped_ptr** για να αποφεύγεται κάποιο memory leak και να υποδηλώνεται το ownership του member από το αντικείμενο.

5.3 Γενικές πρακτικές

1. Για το **iteration** των **STL** containers χρησιμοποιείται όποτε είναι δυνατόν το **BOOST_FOREACH**.
2. Αν παρ'όλα αυτά χρησιμοποιθούν STL iterators το όνομά τους είναι καλά να δείχνει σε τι τύπου container αναφέρεται, π.χ.
   ```
   std::list<int> intList;
   std::list<int>::iterator intListIt;
   ```
3. Οι δηλώσεις εκτέλεσης μέσα στη συνθήκη καλό θα ήταν να αποφεύγονται.

6. Filesystem & ROS packages

6.1. Συμβάσεις στην επέκταση των αρχείων, στα headers και διαχείριση κώδικα

1. Για την ονοματολογία **πακέτων** και **αρχείων**, χρησιμοποιούνται **πεζά** γράμματα και ο διαχωρισμός των λέξεων γίνεται με χρήση **underscore** (π.χ. package_name, file_name.cpp, file_name.h).
2. Η χρήση της επέκτασης .h για headers files ενώ των .cpp για source files.
3. Τα header files έχουν ίδιο όνομα με τα source files (εάν υπάρχουν αντίστοιχα source files).
4. Κάθε κεφαλίδα σε κάθε αρχείο πρέπει να ακολουθεί το παρακάτω πρότυπο:

```
/*********************************************************************
*
* Software License Agreement (BSD License)
*
*  Copyright (c) 2014, P.A.N.D.O.R.A. Team.
*  All rights reserved.
*
*  Redistribution and use in source and binary forms, with or without
*  modification, are permitted provided that the following conditions
*  are met:
*
*   * Redistributions of source code must retain the above copyright
*     notice, this list of conditions and the following disclaimer.
*   * Redistributions in binary form must reproduce the above
*     copyright notice, this list of conditions and the following
*     disclaimer in the documentation and/or other materials provided
*     with the distribution.
*   * Neither the name of the P.A.N.D.O.R.A. Team nor the names of its
*     contributors may be used to endorse or promote products derived
*     from this software without specific prior written permission.
*
*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
*  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
```

```
*  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
*  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
*  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
*  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
*  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
*  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
*  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
*  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
*  POSSIBILITY OF SUCH DAMAGE.
*
* Author: <Name>
*********************************************************************/
```

5. Τα **declarations** των κλάσεων, δηλαδή η δήλωση των ονομάτων, των **μεθόδων** καθώς και των **πεδίων** τους δηλώνονται στο **header** file. Επίσης στο header γίνονται **include** τα απαραίτητα dependencies και ορίζονται τα ενδεχόμενα **typedefs**.
6. Τα **definitions** των μεθόδων των κλάσεων, δηλαδή ο **ορισμός** του **περιεχομένου** τους γίνεται στο αντίστοιχο .cpp.
7. Γενικά καλό είναι να τοποθετείται **μία** κλάση ανά αρχείο. Εξαίρεση ίσως αποτελεί η περίπτωση που οι κλάσεις είναι σχετικά μικρές και έχουν λογική συσχέτιση μεταξύ τους (π.χ. κληρονομικότητα) οπότε θα μπορούσαν να τοποθετούνται στο ίδιο αρχείο.
8. Κάθε **header** file πρέπει να περιλαμβάνει ένα μηχανισμό για την αποφυγή πολλαπλών καταχωρήσεων αρχείων. Αυτό επιτυγχάνεται με τη χρήση ενός **#ifndef/#define** μπλοκ στην αρχή του αρχείου και **#endif** στο τέλος. Δίπλα στο #endif αναγράφεται με σχόλιο πού αναφέρεται. π.χ.

```
#ifndef MY_AWESOME_FILE_H
#define MY_AWESOME_FILE_H

/// declare the classes that do lots
/// of awesome stuff in here

#endif // MY_AWESOME_FILE_H
```

9. Η **σειρά** με την οποία γίνονται τα **include** των dependencies σε ένα **header** είναι η εξής:

   C system files.

   C++ system files.

   Other libraries' .h files.

   ROS libraries' .h files.

   ROS core messages .h files

   Other packages' libraries .h files

Your custom declared message .h files

Your package's .h files.

π.χ.

```
#include <math.h>
#include <vector>
#include <boost/thread>
#include <ros/ros.h>
#include <geometry_msgs/Point.h>
#include "another_ros_package/random_library.h"
#include "my_msgs/MyAwesomeMessage.h"
#include "my_package/my_awesome_file.h"
```

10. Τα ονόματα των αρχείων των **custom επικοινωνιών** που ορίζουμε για το **ROS**, δηλαδή τα **.msg** , **.action** , **.srv** files θα ξεκινούν με **κεφαλαίο** και θα ακολουθούν **mixed-case**, ακριβώς όπως οι κλάσεις.

6.2 Δομή ROS-specific κώδικα

1. Το σύνολο σχεδόν της λειτουργικότητας οργανώνεται σε αντικείμενα. Η **main()** που υλοποιεί ουσιαστικά έναν **κόμβο**, συνήθως **περιορίζεται** στο να κάνει **instantiate** τα απαραίτητα αντικείμενα και να υλοποιεί το **spin**.
2. Τα .cpp αρχεία λοιπόν που περιέχουν τις υλοποιήσεις των μεθόδων, γίνονται **compile** σε **δυναμικές βιβλιοθήκες**, καλώνας το macro **add_library()** στο CMakelists.txt.
3. Η **main()** τοποθετείται σε ξεχωριστό αρχείο .cpp που έχει την κατάληξη _node , ανάλογα με τη λειτουργία που επιτελεί, π.χ. navigation_controller_node.cpp. Το αρχείο αυτό προφανώς κάνει include τα headers που περιέχουν τα αντικείμενα που χρησιμοποιεί. Στο CMakelists.txt γίνεται **compile + link** με το macro **add_executable()** στο οποίο συμπεριλαμβάνονται οι παραπάνω δυναμικές βιβλιοθήκες που χρειάζεται να κάνει **link**.
4. Μια καλή σύμβαση για **top level namespace** στο οποίο ορίζονται οι κλάσεις. είναι να δίνεται το **όνομα του πακέτου.** Προφανώς **δεν** ορίζεται τίποτα ποτέ σε namespace άλλου πακέτου ή βιβλιοθήκης (π.χ. boost::, std:: , ros::).

6.3 Οργάνωση αρχειων σε ros packages

1. Τα headers τοποθετούνται σε ένα directory που έχει **όνομα ίδιο με του πακέτου** και περιέχεται μέσα στο directory **include**, π.χ. στο my_package/include/my_package. Παρατηρήστε ότι το όνομα του πακέτου επαναλαμβάνεται..
2. Τα .cpp αρχεία με τα implementations τοποθετούνται στο directory **src** του πακέτου μας. Εδώ **δεν χρειάζεται** να τοποθετηθεί ακόμα ένα επίπεδο με το όνομα του πακέτου.

7. Αυτόματος έλεγχος συμβάσεων.

Για τον αυτόματο έλεγχο πολλών από τις παραπάνω συμβάσεις στον κώδικα κατά το build θα χρησιμοποιείται το roslint. Βλ. λεπτομέρειες χρήσης στο wiki του github.

8. Αναφορές

1. http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
2. http://wiki.ros.org/CppStyleGuide
3. http://www.doxygen.nl/docblocks.html
4. http://www.doxygen.nl/commands.html#cmdfn (special commands)
5. http://www.learncpp.com/ (cpp tutorial and reference)