Wavelab Coding Standards and Practices

Chris Choi

2016

Disclaimer

- **Programming** is **an art**, not a science
- **Software Engineering** is mostly about the **process of working with other people**

The following content is highly controversial, this talk is meant as a guide to introduce *some* good practices.

A Software Engineer, Programmer and Computer Scientist Walk into a Bar

- **Software Engineer**: Design, Requirements and Collaboration
- **Programmer**: Designs to code, usually works alone.
- **Computer Scientists**: Algorithmist

Ideally you should be all three!

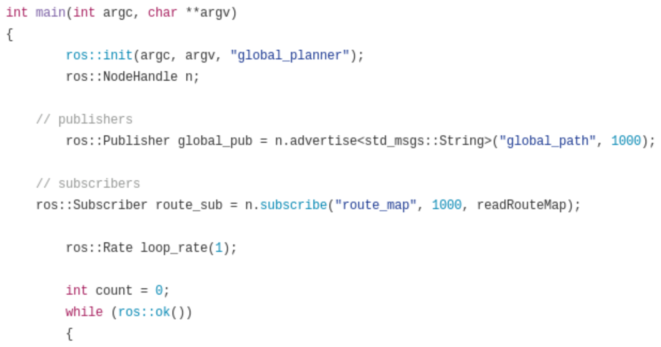
Contents

Indentation

- **Space vs Tabs?**: Should prefer **SPACES** over tabs, this is because an indentation represented by tabs has variable length whereas spaces there is no confusion.
- Research has shown **2 or 4 spaces** is a good indentation level for majority of programmers.
- While you can adjust TAB indentation length with IDEs, we don't all use the same IDE. [1]

[1] I don't recommend we all use the same IDE

Indentation



```
int main(int argc, char **argv)
{
        ros::init(argc, argv, "global_planner");
        ros::NodeHandle n;

    // publishers
        ros::Publisher global_pub = n.advertise<std_msgs::String>("global_path", 1000);

    // subscribers
    ros::Subscriber route_sub = n.subscribe("route_map", 1000, readRouteMap);

        ros::Rate loop_rate(1);

        int count = 0;
        while (ros::ok())
        {
```

Figure : Perfect example why spaces is preferred

Column-Width

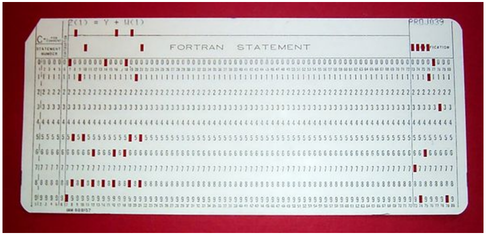


Figure : Fortran Punch Card

Majority of style guides are **80-column width**. This magick number has roots in the good old punch cards where they were 80 column width, and the tradition lives on.

Column-Width

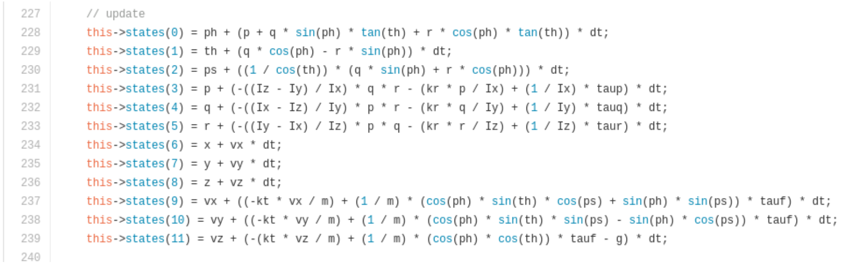
Other valid reasons for narrow margins (80 column width) are:

- To avoid warpping when copying into other media (web pages)
- To view multiple source code side by side
- Increase readibility, and complexity

Exceptions

If wrapping lines means destroying readability, do not wrap.

Column-Width



```
    // update
    this->states(0) = ph + (p + q * sin(ph) * tan(th) + r * cos(ph) * tan(th)) * dt;
    this->states(1) = th + (q * cos(ph) - r * sin(ph)) * dt;
    this->states(2) = ps + ((1 / cos(th)) * (q * sin(ph) + r * cos(ph))) * dt;
    this->states(3) = p + (-((Iz - Iy) / Ix) * q * r - (kr * p / Ix) + (1 / Ix) * taup) * dt;
    this->states(4) = q + (-((Ix - Iz) / Iy) * p * r - (kr * q / Iy) + (1 / Iy) * tauq) * dt;
    this->states(5) = r + (-((Iy - Ix) / Iz) * p * q - (kr * r / Iz) + (1 / Iz) * taur) * dt;
    this->states(6) = x + vx * dt;
    this->states(7) = y + vy * dt;
    this->states(8) = z + vz * dt;
    this->states(9) = vx + ((-kt * vx / m) + (1 / m) * (cos(ph) * sin(th) * cos(ps) + sin(ph) * sin(ps)) * tauf) * dt;
    this->states(10) = vy + ((-kt * vy / m) + (1 / m) * (cos(ph) * sin(th) * sin(ps) - sin(ph) * cos(ps)) * tauf) * dt;
    this->states(11) = vz + (-(kt * vz / m) + (1 / m) * (cos(ph) * cos(th)) * tauf - g) * dt;

```

Figure : 12-State Quadrotor Model

Naming Conventions

General types of naming convention:

- **PascalCase**: capitalize first letter for all words
- **camelCase**: lower first letter, upper first letter for other words
- **snake_case**: all lower, words separated with underscore
- **ALLCAPS**

Apply only 1 type of naming convention at a time to a variable, class or function. NOT a mixture, e.g. "DataBase_Adaptor" (pascal mixed with snake case).

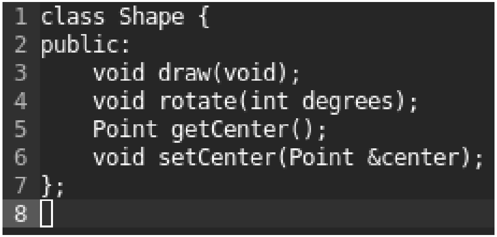
Naming Conventions

- **PascalCase**: Classes
- **camelCase**: Class methods
- **snake_case**: C style functions, variables, member variables, namespaces
- **ALLCAPS**: macros, constants

Note: the above is *debatable* but that is my chosen style [2].

[2] I'll think you're wrong, but I am open to suggestions :P

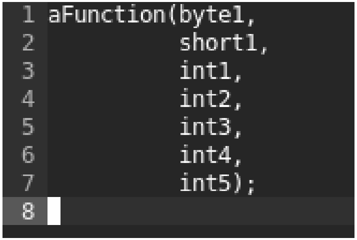
Naming Conventions



```
class Shape {
public:
    void draw(void);
    void rotate(int degrees);
    Point getCenter();
    void setCenter(Point &center);
};
```

Figure : Example Code

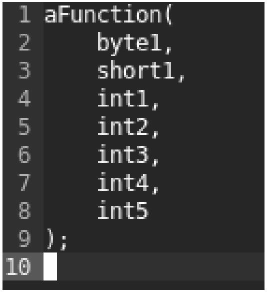
Breaking Multiple Lines



```
aFunction(byte1,
          short1,
          int1,
          int2,
          int3,
          int4,
          int5);
```

Figure : Most Common

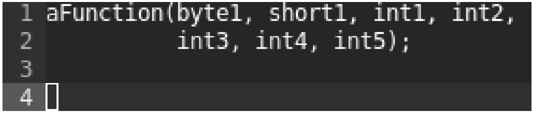
Breaking Multiple Lines



```
aFunction(
        byte1,
        short1,
        int1,
        int2,
        int3,
        int4,
        int5
);
```

Figure : Another Method

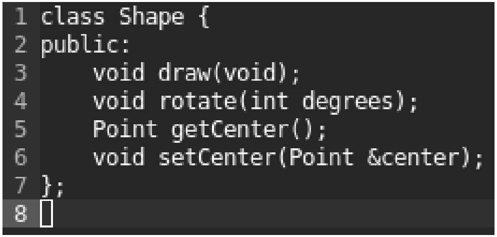
Breaking Multiple Lines



```
aFunction(byte1, short1, int1, int2,
          int3, int4, int5);

```

Figure : NOT RECOMMENDED!

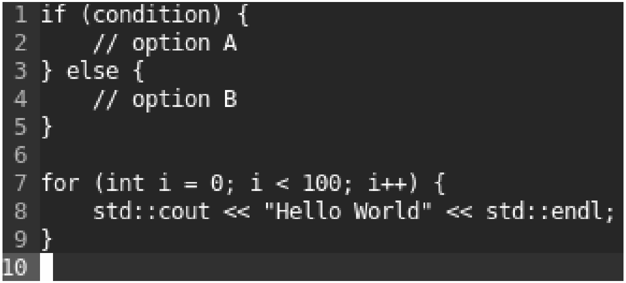
Brace Placement



```
class Shape {
public:
    void draw(void);
    void rotate(int degrees);
    Point getCenter();
    void setCenter(Point &center);
};

```

Figure : Example Code

Brace Placement

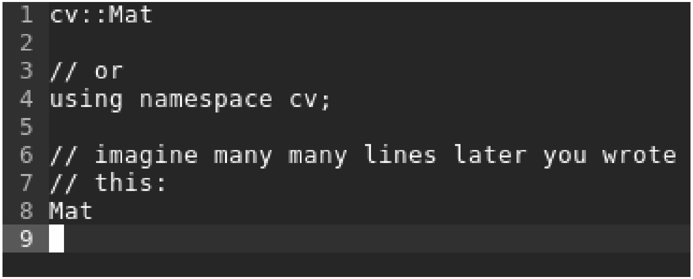


```
if (condition) {
    // option A
} else {
    // option B
}

for (int i = 0; i < 100; i++) {
    std::cout << "Hello World" << std::endl;
}

```

Figure : Example Code

Using Namespace

‘using namespace‘ is generally not a very good idea, you have no idea where the function / class comes from.



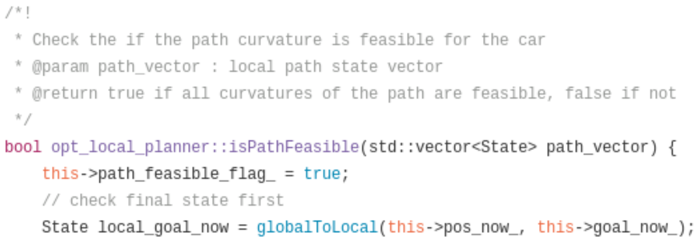
```
cv::Mat

// or
using namespace cv;

// imagine many many lines later you wrote
// this:
Mat

```

Figure : Which one is clearer?

The “this” keyword

```
double PID::calculate(double setpoint, double input, double dt)
{
    double error;
    double output;

    // calculate errors
    error = setpoint - input;
    this->error_sum += error * dt;

    // calculate output
    this->error_p = this->k_p * error;
    this->error_i = this->k_i * this->error_sum;
    this->error_d = this->k_d * (error - this->error_prev) / dt;
    output = this->error_p + this->error_i + this->error_d;

    // update error
    this->error_prev = error;

    return output;
}
```

The “this” keyword



```
/*!
 * Check the if the path curvature is feasible for the car
 * @param path_vector : local path state vector
 * @return true if all curvatures of the path are feasible, false if not
 */
bool opt_local_planner::isPathFeasible(std::vector<State> path_vector) {
    this->path_feasible_flag_ = true;
    // check final state first
    State local_goal_now = globalToLocal(this->pos_now_, this->goal_now_);
```

Figure : Is globalToLocal() a function or class method?

Rationale for ”this”

The idea of using the ”this” keyword is to mitigate confusion, you don’t have to think if you see the ”this” keyword, you know what it means as soon as you see it.

The Myth of the Genius Programmer

Google I/O 2009 - The Myth of the Genius Programmer

Google Developers

982,891

756,071 views

5,344 184

KISS - Keep it Simple Stupid

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." - Antoine de Saint-Exupery

- If you can, prefer the cleaner way of programming
- Avoid over using design patterns
- Keep names short and sweet

Design Patterns

You're already using some probably without knowing:

- Client-Server
- Observer-Subscriber

Design patterns are **general repeatable solutions** to a commonly occurring problem in software design.

Design Patterns

There are many more:

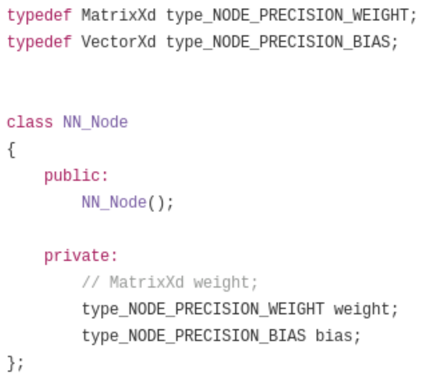
- Factory
- Adaptor
- Singleton
- Model-View-Controller
- And many more

Useful to know when communicating software designs with others.

Warning!

Over use or abuse of design patterns can lead to ANTI-PATTERNS.

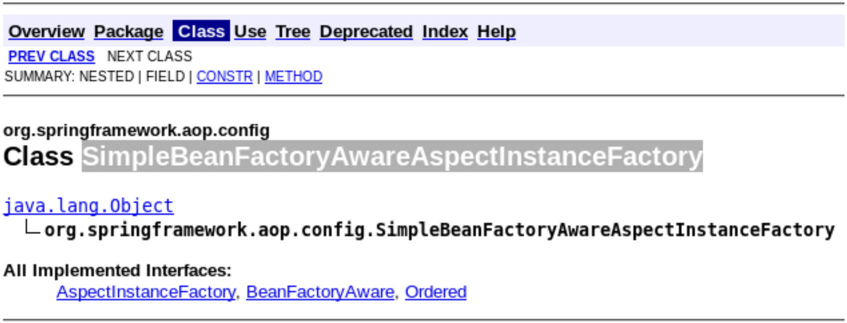
Be Great at Naming Names



```
typedef MatrixXd type_NODE_PRECISION_WEIGHT;
typedef VectorXd type_NODE_PRECISION_BIAS;


class NN_Node
{
    public:
        NN_Node();

    private:
        // MatrixXd weight;
        type_NODE_PRECISION_WEIGHT weight;
        type_NODE_PRECISION_BIAS bias;
};
```

Figure : Nima being silly

Be Great at Naming Names



Overview Package **Class** Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

org.springframework.aop.config
Class SimpleBeanFactoryAwareAspectInstanceFactory

```
java.lang.Object
  └org.springframework.aop.config.SimpleBeanFactoryAwareAspectInstanceFactory
```

All Implemented Interfaces:
AspectInstanceFactory, BeanFactoryAware, Ordered

Figure : Curse of Java

Be Great at Naming Names

Half the battle of programming is coming up with names that other programmers will instantly get. Having good names for classes, functions, variables means **you end up writing less documentation because it is so obvious**.

Follow The Unix Philosophy

Eric Raymonds 17 Unix Rules:

1. Rule of Modularity
2. Rule of Clarity
3. Rule of Composition
4. Rule of Separation
5. Rule of Simplicity
6. Rule of Parsimony
7. Rule of Transparency
8. Rule of Robustness
9. Rule of Representation
10. Rule of Least Surprise
11. Rule of Silence
12. Rule of Repair
13. Rule of Economy
14. Rule of Generation
15. Rule of Optimization
16. Rule of Diversity
17. Rule of Extensibility

Macro-level

Macro-level software development is concerned with the choices you make in regards to the **code architecture**, **testing** and **build processes**.

Test Driven Development (TDD)

1. Write a test to fail
2. Write function to pass test
3. Repeat

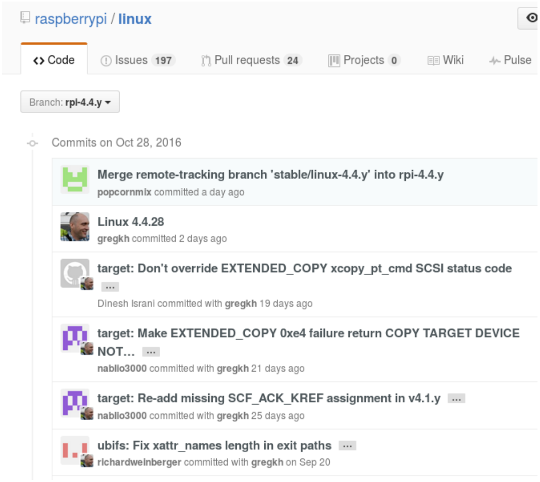
This help forces the programmer think how the function should be called, among other good things ...

Note

Implicitly this means you should have atleast 1 test case per function. Additional testing can now be added trivally.

Commit Messages

Git's default style is to write **present, imperative style** commit messages. E.g.



raspberrypi / **linux**

Commits on Oct 28, 2016

Merge remote-tracking branch 'stable/linux-4.4.y' into rpi-4.4.y
popcornmix committed a day ago

Linux 4.4.28
gregkh committed 2 days ago

target: Don't override EXTENDED_COPY xcopy_pt_cmd SCSI status code ...
Dinesh Israni committed with **gregkh** 19 days ago

target: Make EXTENDED_COPY 0xe4 failure return COPY TARGET DEVICE NOT… ...
nablio3000 committed with **gregkh** 21 days ago

target: Re-add missing SCF_ACK_KREF assignment in v4.1.y ...
nablio3000 committed with **gregkh** 25 days ago

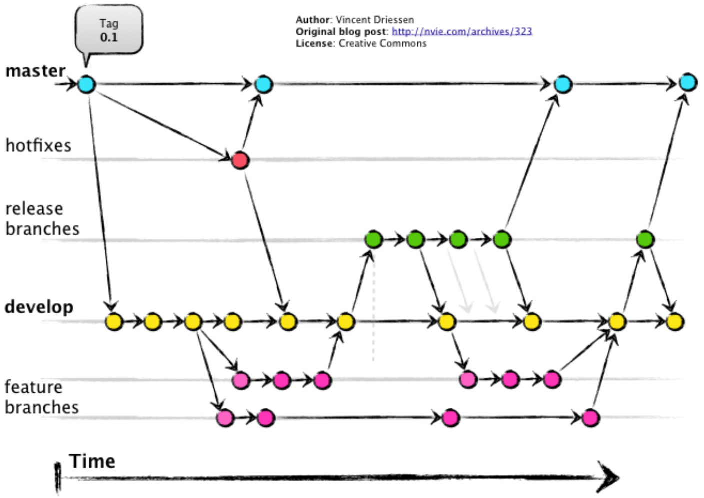
ubifs: Fix xattr_names length in exit paths ...
richardweinberger committed with **gregkh** on Sep 20

Commit Messages

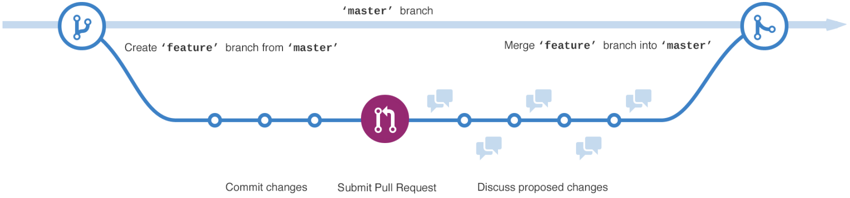
Git is a distributed code versioning tool, commit messages should be framed from the **perspective of the person applying your branch or changes to another branch**, (while many programmers will disagree, using present tense is still a good choice) [3]

[3] if I don't make you do it, Google, Facebook, and others will

Git Flow


Tag
0.1
Author: Vincent Driessen
Original blog post: http://nvie.com/archives/323
License: Creative Commons
master
hotfixes
release branches
develop
feature branches
Time

GitHub Flow


'master' branch
Create 'feature' branch from 'master'
Merge 'feature' branch into 'master'
Commit changes
Submit Pull Request
Discuss proposed changes

GitHub Flow

For our sanity I think I know what I'll choose. But I would like to add 1 thing to the GitHub flow process. (I did not invent this, its pretty standard practice)

One Issue Per Branch

Treat the dedicated issue for the branch as a requirements / specification doc. Serves as a place to discuss requirements.

GitHub Flow - Modified

Create Issue BEFORE branching

Issue should describe the context, as well as an insight of how you're doing it. (e.g. What are you doing? How are you doing it; A requirements document)

Create Branch

Branch from master, using simple naming conventions "fix-31", "feature-12", with number indicating the corresponding issue number it is linked to.

Finish Development Create Pull Request

Once you finished development, create Pull Request for code review.

Merge and Delete Branch

Easy Peasy Lemon Squeezy

There are many more topics I haven't covered, but what I have discussed is more than enough for a single talk. You'll just have to learn on the job :)

The End!