MAS Group Coding Conventions
Version 0.1.0

Matthias Füller, Frederik Hegger, Nico Hochgeschwender, Sven Schneider

June 26, 2014

1 Generic hints

- The overall goal to write easily *readable* code. Thus, in certain situations it may be preferable to deviate from these conventions.
- When working on existing code, always stay consistent with the already applied convention!

  Rationale: Mixing of different coding styles/conventions results in hardly readable code.
- Don't check-in commented code.

  Rationale: To keep track of changes in code, we have the version control system. Additionally, it is unclear what the meaning of commented code is (maybe it is not working, not tested or contains a bug).

2 Python

- Code: *Python Enhancement Proposal (PEP) 8 – Style Guide for Python Code*[1]
- Documentation: *Python Enhancement Proposal (PEP) 8 – Style Guide for Python Code*[2]

[1] `http://legacy.python.org/dev/peps/pep-0008/`
[2] `http://legacy.python.org/dev/peps/pep-0008/`

3 Java

- Code: *Code Conventions for the Java Programming Language*[3]
- Documentation: *JavaDoc*[4]

[3] http://www.oracle.com/technetwork/java/codeconv-138413.html

[4] http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html

4 C++

4.1 Code Conventions

4.1.1 Indentation

- Use spaces for indentation.

  Rationale: Tabs may be configured to have different width and thus lead to inconsistencies.

- Each level is indented by four spaces.

  Rationale: Especially in more deeply nested code two spaces of indentation are not sufficient, while eight spaces of indentation are too much.

- Indent the following statements by one level:

  - The access modifiers (public, protected, private) in classes
  - The declarations within the access modifiers (public, protected, private)
  - Statements within blocks (blocks are e.g.: function bodies, *switch* bodies or *case* bodies)
  - Labels
  - Pre-processor definitions which end in a backslash
  - Comments which are placed on the first line of a new scope

  Rationale: These indentations help to structure the source code hierarchically.

- Do **not** indent the following statements:

  - *break* statements within *case* or *default*, if no explicit scope is provided by braces
  - Declarations within *namespace* definitions
  - Empty lines

Rationale: The *break* statement limits the "scope" of a *case* and thus should be easily visible. Namespace definitions are "rare" and therefore should not waste one indentation level. Indentation of empty lines is hardly visible and therefore tends to accumulate.

Listing 1: Example *Indentation* in a namespace and class declaration

```
#define␣SQUARE(X)␣\
␣␣␣␣(X)␣*␣(X)

namespace␣foobar
{

class␣Foo
{
␣␣␣␣public:
␣␣␣␣␣␣␣␣Foo();
␣␣␣␣␣␣␣␣void␣foo();

␣␣␣␣private:
␣␣␣␣␣␣␣␣int␣bar_;
}

}
```

Listing 2: Example *Indentation* of blocks in a function

```
void␣foo(int␣bar,␣int␣x,␣int␣y)
{
␣␣␣␣//␣A␣comment␣in␣the␣first␣line␣of␣a␣scope
␣␣␣␣switch␣(bar)␣{
␣␣␣␣␣␣␣␣case␣0:
␣␣␣␣␣␣␣␣␣␣␣␣++bar;
␣␣␣␣␣␣␣␣break;
␣␣␣␣␣␣␣␣case␣1:
␣␣␣␣␣␣␣␣␣␣␣␣--bar;
␣␣␣␣␣␣␣␣default:␣{
␣␣␣␣␣␣␣␣␣␣␣␣bar␣+=␣bar;
␣␣␣␣␣␣␣␣␣␣␣␣break;
␣␣␣␣␣␣␣␣}
␣␣␣␣}

␣␣␣␣if␣(x␣<␣y)␣{
␣␣␣␣␣␣␣␣return␣-1;
␣␣␣␣}␣else␣if␣(x␣>␣y)␣{
```

```
        return 1;
    } else {
        return 0;
    }
}
```

4.1.2 Line Wrapping

- The maximum length of a line is 120 characters. Longer lines must be wrapped.

  Rationale: Screen sizes have increased significantly since the time of 80 character terminals and terminal emulators usually can be resized. 120 characters per line allow for a more efficient use of bigger screens while still not overcrowding individual lines.

- Indent wrapped lines by two levels.

  Rationale: This easily separates the wrapped line from the following statement (which in turn is indented by only one level).

Listing 3: Example *Line Wrapping*

```
class Example : public FooClass,
        virtual protected BarClass
{
};
```

4.1.3 Braces: One true brace style (1TBS)

- For namespaces, classes and functions the opening brace is placed on the next line with the same indentation level as the header.

  Rationale: In case of wrapped lines, e.g. due to many arguments to a function or multiple inheritance, the opening brace stays easily visible.

- For control statements, the opening brace is placed on the same line as the control statement.

Rationale: This style does not waste space. Additionally, statements should be kept short. Thus, the brace is easier to find.

- In *if/else* statements, if the consequent or alternative is only one line of code, it can be placed on the same line as the *if/else* statement (the combination of both lines has to stay within the maximum line length, though).

  Rationale: Usually, this style is used in a block of "short" checks e.g. to validate input parameters. In this case no space is wasted by the two additional lines (consequent/alternative + brace). This also allows the "safe" insertion of new code lines.

- The closing brace is always placed on a separate line, except when followed by an *else* or *while*.

  Rationale: This allows to easily identify the end of a scope.

Listing 4: Example of the *One true brace style (1TBS)*

```
int␣foo(bool␣is_bar,␣void␣*bar)
{
␣␣␣␣if␣(!bar)␣return␣0;

␣␣␣␣if␣(is_bar)␣{
␣␣␣␣␣␣␣␣bar();
␣␣␣␣␣␣␣␣return␣1;
␣␣␣␣}␣else␣{
␣␣␣␣␣␣␣␣return␣0;
␣␣␣␣}
}
```

4.1.4 Padding

- Spaces should be placed:
  - After keywords
  - After commas
  - After semicolons, if a statement follows
  - Before and after colons in base clauses and the ternary operator

  - Before opening braces
  - After closing braces, if a statement follows
  - Before and after binary operators (assignment, mathematical or logical operators)
  - After colons of label declarations
  - Before and after the question mark of the ternary operator

- Spaces should **not** be placed:
  - After function names
  - After opening parantheses, brackets and angle brackets
  - Before closing parantheses, brackets and angle brackets
  - Before commas and semicolons
  - Before colons of label declarations and *case* or *default* statements
  - After prefix or before postfix operators
  - After unary operators
  - Before the angle brackets of templates

Listing 5: Example of the padding. The "+" in the beginning of a comment line means "add a space", while the "-" means "don't add a space".

```
// + Around base clause colon
class Foo : Bar
{
    public:
        // + After comma
        // - After function name
        // - After opening paranthesis
        // - Before closing paranthesis
        // - Before comma
        char foo(bool is_bar, int bar)
        {
            // + Around colon/question mark in ternary operator
            // - Before semicolon
            int val = (bar == 0) ? 0 : 1;

            // + After keyword
            if (is_bar) {
```

```
                //␣-␣After␣function␣name
                return␣fooBar();

            //␣+␣After␣closing␣brace␣and␣before␣opening␣brace
            }␣else␣{
                //␣-␣Before␣postfix␣operator
                val++;
            }

            //␣+␣After␣semicolon
            for␣(int␣i␣=␣0;␣i␣<␣42;␣i++)␣{
                //␣+␣Around␣binary␣operators
                val␣=␣val␣*␣2;
            }

            switch␣(val)␣{
                //␣-␣Before␣colon␣of␣case␣statement
                case␣0:
                    //␣-␣After␣unary␣operator
                    val␣=␣-1;
                break;

                default:
                    val␣=␣42;
            }

            //␣-␣Before␣the␣angle␣brackets␣of␣a␣template
            return␣static_cast<char>(val);
        }
};
```

4.1.5 Pointer alignment

- In the declaration of pointers or references, attach the pointer (*) or reference (&) operator to the name.

  Rationale: A declaration like `int* x, y;` declares one integer pointer x and one integer y. When the pointer operator is attached to the name, this is more clearly visible.

Listing 6: Example of the *Pointer alignment*

```
int␣*x,␣y;
```

4.1.6 Naming

- This is a list of used naming styles:
  - Upper CamelCase: In compound words, the first letter of each element/word is a capital letter. The compound word starts with an upper-case letter. For example: FooBar
  - Lower CamelCase: In compound words, the first letter of each element/word is a capital letter. The compound word starts with a lower-case letter. For example: fooBar
  - snake_case: Only lower case letters are used. Compound words are separated by an underscore. For example: foo_bar
  - SCREAMING_SNAKE_CASE: Only upper case letters are used. Compound words are separated by an underscore. For example: FOO_BAR
- Use the following naming conventions:
  - Classes and enumerations: Upper CamelCase
  - Files: snake_case, same as class name
  - Local and global variables: snake_case
  - Member variables: snake_case, with trailing underscore
  - Function name: Lower CamelCase
  - Constant: SCREAMING_SNAKE_CASE
  - Enumeration literals: SCREAMING_SNAKE_CASE
  - Preprocessor definitions: SCREAMING_SNAKE_CASE
- Preprocessor include guards should be named after the file name (in SCREAMING_SNAKE_CASE) followed by "_HPP".
- Don't start variable names or preprocessor definitions with one or more underscores.

  Rationale: Certain names beginning with one underscore and all names beginning with two underscores are reserved for the compiler.

4.1.7 File structure

- Create two files per class, one for the declarations (*.hpp) and one for the implementations (*.cpp).
- Optionally, when templates are used, add another file (*.impl) for template implementations. This file is then included at the end of the respective header file.

4.2 Documentation Conventions

- Use a Doxygen-compatible style for documenting `C++` code. Note: Doxygen is also able to interpret JavaDoc style.

4.3 Best practices

- Preprocessor include directives: Double quotes for includes relative to the location of the current file. Angle brackets for other locations.
- Always declare the default constructor, copy constructor, destructor and assignment operator explicitly. The copy constructor and the assignment operator can initially be declared as private and only be implemented when required.

  Rationale: In many situations the auto-generated versions of these functions do not have the desired or expected behavior.
- Declare only one variable per line.

  Rationale: This prevents the problems arising from pointer alignment and makes variable declarations easier to find.
- Write const-correct code.
- Write exception-safe code.
- Don't leak resources (e.g. memory, file handles, ...).

5 ROS packages

5.1 package.xml

- Versioning: *Semantic Versioning 2.0.0*[5]
- License: GPLv3
- Maintainer and author email: Your university email address
- Keep the dependencies minimal and only include dependencies if and when they are required.
- Use spaces for indentation.
- Each level is indented by two spaces.

5.2 CMakeLists.txt

- Use spaces for indentation.
- Each level is indented by two spaces.
- Line wrapping: Place closing paranthesis on the same indentation level as the opening command.
- If there is a list of "named arguments" (e.g. `COMPONENTS abc def`, where `COMPONENT` is the name and `abc def` are the arguments), place the name on a separate line and then each argument on a separate line, indented by one level w.r.t. to the name.
- Omit optional or non-required arguments or keywords.

5.3 Launch files

- Use spaces for indentation.
- Each level is indented by four spaces.

[5] `http://semver.org/spec/v2.0.0.html`