Source code

Header files

Extension

Use *.h as file extension.

Include guards

Use the simplified include guard as the first line of the header file:

```
#pragma once
```

The old style include guard ( #ifndef __YOURFILE_H__ ) is harder to maintain, e.g. when file names are changed.

For more information, see [https://en.wikipedia.org/wiki/Pragma_once](https://en.wikipedia.org/wiki/Pragma_once)

Source files

Extension

Use *.cpp as file extension.

General

Naming convention

```
/* FooBar.cpp */  // File: UpperCamelCase

class FooBar       // Class: UpperCamelCase
{

private:
```

THIS RULES ARE STILL UNDER DISCUSSION
static constexpr int MAGIC_NUMBER {-999};       // Constants: UPPER_CASE
int myMember;                                    // Members: lowerCamelCase
FooBar();                                        // Ctor: UpperCamelCase

public:
void Bar();                                      // Methods: UpperCamelCase
void BarBar(bool flag, int counter);            // Arguments: lowerCamelCase
void TbdBar(); // Tbd = To Be Discussed */        // Abbreviations: UpperCamelCase

- Abbreviations used in files/classes/methods/variables are used like words and follow the UpperCamelCase rule.
  Examples:
  - AgentID → AgentId
  - ADASDriver → AdasDriver
- Avoid public class data members. If unavoidable, use lowerCamelCase.
- Avoid global variables or at least try to make them constants. Nevertheless (following the google c++ guide), all global variables (or constants) should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global. Use lowerCamelCase.
- Enums names are treated like class names (especially since enum class is preferred over a plain enum): UpperCamelCase
- Do not use Hungarian notation ( iCounter → counter ),
- Similar: Avoid specifying the type of the underlying container ( partMap → parts ).
- Do not use magic numbers in the code. Make it explicit as a constant.
- Use slashes // as comment type, avoid /* */

**Enforcing the rules**

TO BE COMPLETED

*we currently thinking about an adjustment of google's cpplint.py for this purpose*

**Documentation**

TO BE COMPLETED
we use doxygen

Polymorphthic Methods

Do not comment on polymorph methods (virtual base → override), unless there is a severe change.

Code Formatting

From google cppguide:

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone’s code easily.

Tooling

To help you format code correctly, the following tools and settings are useful

- **Code formatter**
  Artistic style (aka `astyle`)
  - Install on linux: `sudo apt install astyle`
  - Install on windows: Download on sourceforge

- **IDE Integration**
  QT Creator Beatuifier Plugin
  - Enable via Help → About Plugins → C++:Beautifier (experimental)
  - Setup via Tools → Options → Beautifier
  - Recommended options:
    - **General**
      → Enable auto format on save: be careful what you wish for
      → Tool: artistic style
    - **Artistic style**
      → Artistic Style command: depending on your install path
      → Use customized style: see predefined style set below
• Manual use (if not applied on save): Tools → Beautifier Artistic Style → Format current file

• **Useful stuff**

  QT Project Tools (qpt)

  • Collection of command line tools.
    
    Allows too apply astyle to to a QT Project (*.pro)
  
  • Install on linux: Build from source (see sourceforge)
  
  • Install on windows: Download on sourceforge
  
  • Configure qpt.ini section [astyle]: see predefined style set below
  
  • Run with qpt -m /pathToQMake/qmake -t astyle yourproject.pro

---

**Predefined style set**

*THESE RULES ARE STILL UNDER DISCUSSION*

Please refer to the [astyle documentation](#) for the individual rules.

```bash
--style=allman
--indent=spaces=4
--attach-namespaces
--attach-closing-while
--indent-switches
--indent-preproc-cond
--indent-coll-comments
--max-continuation-indent=60
--pad-oper
--pad-comma
--pad-header
--unpad-paren
--align-pointer=type
--align-reference=type
--break-closing-braces
--add-braces
--keep-one-line-blocks
--convert-tabs
```
Code Quality Checks

Clang Code Model Checks

*THESE RULES ARE STILL UNDER DISCUSSION*


In our project use the following tools (and settings)

- Clang Code Model:

  ```
  ```

- Clang-Tidy Checks:

  ```
  -*,bugprone-*,cppcoreguidelines-*,misc-*,modernize-*,performance-*,readability-*
  ```

- Clazy tests:

  **Level 0: no false positives**

Note, that this option can slow down your machine. If so, you can still apply the checks manually.

Static Code Analysis

*THESE RULES ARE STILL UNDER DISCUSSION*
- **Analysis Tool**
  CppCheck
  - Install instructions (linux/windows): http://cppcheck.sourceforge.net/

- **IDE Integration**
  QT Creator QtCppcheck Plugin
  - Enable via `Help → About Plugins → C+QtCppcheck`
  - Setup via Tools → Options → Analyser → CppCheck
  - Recommended options:
    - Custom Parameters: `--std=c++11 --language=c++`
    - Optional: Check on Build *(this may slow down your build)*
    - Optional: Check for inconclusive errors *(this might detect false positives)*
  - Execute: Tools → CppCheck → Check current project

- **Usefull stuff**
  QT Project Tools *(qpt)*
  - Collection of command line tools.
    - Allows too apply cppcheck to to a QT Project (*.pro)
  - Install on linux: Build from source *(see sourceforge)*
  - Install on windows: Download on sourceforge
  - Adjust qpt.ini section [cppcheck]:
    ```
    Add_project_includes_using_dash_I__VALUES__false_true=false
    ```
  - Run with `qpt -m /pathToQMake/qmake -t cppcheck --std=c++11 --language=c++ yourproject.pro`