OpenPASS Conventions

General

OpenPASS is based on modern C++ (currently C++17). For coding guidelines, please refer to ISO C++ Core Guidelines

Headers/Sources

- Use `*.h` as file extension for header files
- Use `*.cpp` as file extension for source files

Naming Conventions

Concise Summarized Naming Conventions Example

```
#pragma once

namespace openpass::component::algorithm
{

/* fooBar.h */                                          // File:  to be discussed
class FooBar                                            // Class: UpperCamelCase
{
private:
    static constexpr int MAGIC_NUMBER {-999};           // Constants: UPPER_CASE
    int myMember;                                       // Members:
lowerCamelCase
    FooBar();                                           // Ctor:
UpperCamelCase

    openpass::component::common::Ports inputPorts;      // Inputs of the class if
used as model
    openpass::component::common::Ports outputPorts;     // Outputs of the class if
used as model

public:
    void Bar();                                         // Methods:
UpperCamelCase
    void BarBar(bool flag, int counter);                // Arguments:
lowerCamelCase
    void YaaBar(); /* Yaa = Yet Another Abbreviation */ // Abbreviations:
UpperCamelCase
};

}
```

Namespaces

- Use lowercase for namespaces
- Use singular form for namespaces where appropriate
- Use base namespace `openpass`
- Core uses `openpass::core::*`
- Components use `openpass::component::*`
- Use the appropriate namespace for the type your component:
  - `openpass::component::algorithm`
  - `openpass::component::sensor`
  - `openpass::component::dynamics`
  - `openpass::component::driver`
  - ...
- Code with shared scope (e.g. `common`) namespaces are separated in:
  - For everyone `openpass::common`: e.g. `openpass::common::XmlParser`
  - Common for components `openpass::component::common`: e.g. `openpass::components::Ports`
  - For the core only `openpass::core::common`: e.g. `openpass::core::common::Parameters`
- **Discussion:** `openpass::type::*`
  Example: `openpass::type::Vector2D`, `openpass::type::OpenDriveId`

Classes

- Classes should be named descriptively according to the functionality they implement with an `UpperCamelCase` name
- A Class implementing an Interface should have the Interfaces name (see below), with the `Interface` portion removed
  Example: `class AgentBlueprint : public AgentBlueprintInterface {...};`

Methods

- Methods should be descriptively named in `UpperCamelCase`
  Example: Method for retrieving the time of day should be named `GetTimeOfDay()`

Member Variables

- Member variables should be descriptively named in `lowerCamelCase`
- Normally, it is sufficient to use the classes name directly:
  Example: The member variable containing the AgentNetwork should be named `agentNetwork`

Input / Output Signal Naming

- Components use a special form of signal transmission. For easier use, the following abstraction is recommended:

- `std::map<int, ComponentPort *> outputPorts;`
  `bool success = outputPorts.at(localLinkId)->SetSignalValue(data);`

- `std::map<int, ComponentPort *> inputPorts;`
  `bool success = inputPorts.at(localLinkId)->GetSignalValue(data);`

- **Discussion:** Wrap in `openpass::components::common::Port` and further `openpass::components::common::Ports`

  ```
  namespace openpass::component::common
  {
      class Port {... };
      using Ports = std::map<int, Port *> Ports;
  }
  ```

Additional Stuff

- Use `UpperCamelCase` for abbreviations used in files, classes, methods, or variables
- This does not apply if the abbreviation is the entire name or the beginning of the name - in such a case the name is written with the rules for the appropriate type
  - `int ID`→`int id`
  - `class AgentID`→ `class AgentId`
  - `ADASDriver.cpp`→`adasDriver.cpp`
- Use `UPPER_SNAKE_CASE` (and `constexpr`) for all constants
- Enums should be preferably defined as enum class; as such, enum names should be in `UpperCamelCase`
- Decorate container by type aliases and use `UpperCamelCase`:
  Example: `using FooParts = std::vector<FooPart>;`
- Use `//` for comments

Avoid

- Do **not** use Hungarian notation for variables names (`iCounter`→`counter`)
- Do **not** specify the type of the underlying implementation (`partMap`→`parts`)
- Do **not** use magic numbers in the code; explicitly define constants instead
- Do **not** use `/* */` for comments
- Do **not** use global variables

Exceptions

- Autogenerated code does not need to follow the coding conventions
  Example: Signals/Slots (QT): `void on_button_clicked();`

Formatting

- A `.clang-format` file is provided at the root level

- It is recommended to auto-format the files on save (see Beautifier Plugin)

- Note, we aim for auto-formatting commits for better comparability.
- **Proposal:**

  ```
  BasedOnStyle: llvm
  Language: Cpp
  ColumnLimit: 0
  IndentWidth: 4
  AccessModifierOffset: -4
  IncludeBlocks: Regroup
  IncludeCategories:
    - Regex:           '^<(gtest|gmock)/)'
      Priority:        -1
    - Regex:           '^<[^Q]'
      Priority:        1
    - Regex:           '^<Q'
      Priority:        2
  AlignTrailingComments: true
  BreakConstructorInitializers: AfterColon
  ConstructorInitializerAllOnOneLineOrOnePerLine: true
  AllowShortFunctionsOnASingleLine: None
  KeepEmptyLinesAtTheStartOfBlocks: false
  BreakBeforeBraces: Custom
  BraceWrapping:
    AfterClass:            true
    AfterControlStatement: true
    AfterEnum:             true
    AfterFunction:         true
    AfterNamespace:        false
    AfterObjCDeclaration:  true
    AfterStruct:           true
    AfterUnion:            true
    AfterExternBlock:      true
    BeforeCatch:           true
    BeforeElse:            true
    IndentBraces:          false
    SplitEmptyFunction:    true
    SplitEmptyRecord:      true
    SplitEmptyNamespace:   true
  ForEachMacros:    [ foreach, Q_FOREACH, BOOST_FOREACH, forever, Q_FOREVER,
  QBENCHMARK, QBENCHMARK_ONCE ]
  ```

Coding Conventions

Interfaces

- Interfaces should be named descriptively according to the functionality they outline with an `UpperCamelCase` name
  Example: Interface for the **world** = `class WorldInterface`
- Interfaces are abstract classes, and as such provide pure virtual functions only, withtou any default implementation.

  Exmaple: `virtual double GetDistance() const = 0;`
- Interface methods **do not** exibit default parameters.
- We excessively use **gmock**, so for every interface a fake interface should be provided
  Example: `class FakeWorld : public WorldInterface {...};`
  Note: Following *Roy Osherove*, we use Fake instead of Mock, whick allows to distinguish Mocks and Stubs more easily in the code.

Documention

- Use Doxygen for documentation
- As Doxygen automatically populates the documentation of base class methods to derived ones, **do not** document derived methods, unless there is a good reason to do so.

End Of Line

- Use linux line endings
- Recommendations:
  - Under windows add `git config --local core.autocrlf true` to your `.gitconfig` file
  - Under linux add `git config --local core.autocrlf input` to your `.gitconfig` file