This document specifies how to contribute code to CAF.

Git Workflow

============

Please adhere to the following Git naming and commit message conventions.

Having a consistent work flow and style reduces friction and makes organizing

contributions a lot easier for all sides.

Branches

--------

- Our main branch is `master`. It reflects the latest development changes for

the next release and should always compile. Nightly versions use the

`master`. Users looking for a production-ready state are encouraged to use

the latest release version instead.

- Push trivial bugfixes (e.g. typos, missing includes, etc.) consisting of a

single commit directly to `master`. Otherwise, implement your changes in a

topic or bugfix branch and file a pull request on GitHub.

- Implement new features and non-trivial changes in a \*topic branch\* with

naming convention `topic/short-description`.

- Implement fixes for existing issues in a \*bugfix branch\* with naming

convention `issue/$num`, where `$num` is the issue ID on GitHub.

- Simply use a fork of CAF if you are an external contributor.

Pull Requests

-------------

Check the following steps to prepare for a merge into `master` after completing

work in a topic or bugfix branch (or fork).

- Squash your commits into a single one if necessary. Each commit should

represent a coherent set of changes.

- Wait for a code review and the test results of our CI.

- Address any review feedback and fix all issues reported by the CI.

- A maintainer will merge the pull request when all issues are resolved.

Commit Message Style

--------------------

- Summarize the changes in no more than 50 characters in the first line.

Capitalize and write in an imperative present tense, e.g., "Fix bug" as

opposed to "Fixes bug" or "Fixed bug".

- Suppress the dot at the end of the first line. Think of it as the header of

the following description.

- Leave the second line empty.

- Optionally add a long description written in full sentences beginning on the

third line. Indent at 72 characters per line.

Coding Style

============

When contributing source code, please adhere to the following coding style,

which is loosely based on the [Google C++ Style

Guide](https://google.github.io/styleguide/cppguide.html) and the coding

conventions used by the C++ Standard Library.

Example for the Impatient

-------------------------

```c++

// libcaf\_example/caf/example/my\_class.hpp

#pragma once

#include <string>

// use "//" for regular comments and "///" for doxygen

namespace caf {

namespace example {

/// This class is only being used as style guide example.

class my\_class {

public:

/// Brief description. More description. Note that CAF uses the

/// "JavaDoc-style" autobrief option, i.e., everything up until the

/// first dot is the brief description.

my\_class();

/// Destructs `my\_class`. Please use Markdown in comments.

~my\_class();

// suppress redundant @return if you start the brief description with "Returns"

/// Returns the name of this instance.

inline const std::string& name() const noexcept {

return name\_;

}

/// Sets the name of this instance.

inline void name(const std::string& new\_name) {

name\_ = new\_name;

}

/// Prints the name to `std::cout`.

void print\_name() const;

/// Does something (maybe).

void do\_something();

/// Does something else but is guaranteed to never throw.

void do\_something\_else() noexcept;

private:

std::string name\_;

};

} // namespace example

} // namespace caf

```

```c++

// libcaf\_example/src/example/my\_class.cpp

#include "caf/example/my\_class.hpp"

#include <iostream>

namespace caf {

namespace example {

namespace {

constexpr const char default\_name[] = "my object";

} // namespace

my\_class::my\_class() : name\_(default\_name) {

// nop

}

my\_class::~my\_class() {

// nop

}

void my\_class::print\_name() const {

std::cout << name() << std::endl;

}

void my\_class::do\_something() {

if (name() == default\_name) {

std::cout << "You didn't gave me a proper name, so I "

<< "refuse to do something."

<< std::endl;

} else {

std::cout << "You gave me the name \"" << name()

<< "\"... Do you really think I'm willing to do something "

"for you after insulting me like that?"

<< std::endl;

}

}

void my\_class::do\_something\_else() noexcept {

// Do nothing if we don't have a name.

if (name().empty())

return;

switch (name.front()) {

case 'a':

// handle a

break;

case 'b':

// handle b

break;

default:

handle\_default();

}

}

} // namespace example

} // namespace caf

```

```c++

// libcaf\_example/test/example/my\_class.cpp

#define CAF\_SUITE example.my\_class // name of this test suite

#include "caf/example/my\_class.hpp" // header-under-test

#include "caf/test/dsl.hpp" // caf::test includes

#include <iostream> // standard includes

// ... // other CAF includes

namespace {

struct fixture {};

} // namespace

CAF\_TEST\_FIXTURE\_SCOPE(my\_class\_tests, fixture)

// ... any number of CAF\_TEST ...

CAF\_TEST\_FIXTURE\_SCOPE\_END()

```

General

-------

- Use 2 spaces per indentation level.

- Use at most 80 characters per line.

- Never use tabs.

- Never use C-style casts.

- Never declare more than one variable per line.

- Only separate functions with vertical whitespaces and use comments to

document logical blocks inside functions.

- Use `.hpp` as suffix for header files and `.cpp` as suffix for implementation

files.

- Bind `\*` and `&` to the \*type\*, e.g., `const std::string& arg`.

- Never increase the indentation level for namespaces and access modifiers.

- Use the order `public`, `protected`, and then `private` in classes.

- Always use `auto` to declare a variable unless you cannot initialize it

immediately or if you actually want a type conversion. In the latter case,

provide a comment why this conversion is necessary.

- Never use unwrapped, manual resource management such as `new` and `delete`.

- Prefer `using T = X` over `typedef X T`.

- Insert a whitespaces after keywords: `if (...)`, `template <...>`,

`while (...)`, etc.

- Put opening braces on the same line:

```c++

void foo() {

// ...

}

```

- Use standard order for readability: C standard libraries, C++ standard

libraries, OS-specific headers (usually guarded by `ifdef`), other libraries,

and finally (your) CAF headers. Include `caf/config.hpp` before the standard

headers if you need to include platform-dependent headers. Use angle brackets

for system includes and doublequotes otherwise.

```c++

// example.hpp

#include <vector>

#include <sys/types.h>

#include "3rd/party.h"

#include "caf/fwd.hpp"

```

Put the implemented header always first in a `.cpp` file.

```c++

// example.cpp

#include "caf/example.hpp" // header for this .cpp file

#include "caf/config.hpp" // needed for #ifdef guards

#include <algorithm>

#ifdef CAF\_WINDOWS

#include <windows.h>

#else

#include <sys/socket.h>

#endif

#include "some/other/library.h"

#include "caf/actor.hpp"

```

- Put output parameters in functions before input parameters if unavoidable.

This follows the parameter order from the STL.

- Protect single-argument constructors with `explicit` to avoid implicit

conversions.

- Use `noexcept` whenever it makes sense and as long as it does not limit future

design space. Move construction and assignment are natural candidates for

`noexcept`.

Naming

------

- All names except macros and template parameters should be

lower case and delimited by underscores.

- Template parameter names should be written in CamelCase.

- Types and variables should be nouns, while functions performing an action

should be "command" verbs. Classes used to implement metaprogramming

functions also should use verbs, e.g., `remove\_const`.

- Private and protected member variables use the suffix `\_` while getter \*and\*

setter functions use the name without suffix:

```c++

class person {

public:

person(std::string name) : name\_(std::move(name)) {

// nop

}

const std::string& name() const {

return name\_

}

void name(const std::string& new\_name) {

name\_ = new\_name;

}

private:

std::string name\_;

};

```

- Use `T` for generic, unconstrained template parameters and `x`

for generic function arguments. Suffix both with `s` for

template parameter packs and lists:

```c++

template <class... Ts>

void print(const Ts&... xs) {

// ...

}

void print(const std::vector<T>& xs) {

// ...

}

```

Headers

-------

- Each `.cpp` file has an associated `.hpp` file.

Exceptions to this rule are unit tests and `main.cpp` files.

- Each class has its own pair of header and implementation files and the

relative path for the files are derived from the full class name. For

example, the header file for `caf::example::my\_class` of `libcaf\_example` is

located at `libcaf\_example/caf/example/my\_class.hpp` and the source file at

`libcaf\_example/src/example/my\_class.cpp`.

- All header files use `#pragma once` to prevent multiple inclusion.

- Do not `#include` when a forward declaration suffices.

- Each library component must provide a `fwd.hpp` header providing forward

declarations for all types used in the user API.

- Each library component should provide an `all.hpp` header that contains the

main page for the documentation and includes all headers for the user API.

- Use `inline` for small functions (rule of thumb: 10 lines or less).

Breaking Statements

-------------------

- Break constructor initializers after the comma, use two spaces for

indentation, and place each initializer on its own line (unless you don't

need to break at all):

```c++

my\_class::my\_class()

: my\_base\_class(some\_function()),

greeting\_("Hello there! This is my\_class!"),

some\_bool\_flag\_(false) {

// ok

}

other\_class::other\_class() : name\_("tommy"), buddy\_("michael") {

// ok

}

```

- Break function arguments after the comma for both declaration and invocation:

```c++

intptr\_t channel::compare(const abstract\_channel\* lhs,

const abstract\_channel\* rhs) {

// ...

}

```

- Break before tenary operators and before binary operators:

```c++

if (today\_is\_a\_sunny\_day()

&& it\_is\_not\_too\_hot\_to\_go\_swimming()) {

// ...

}

```

Template Metaprogramming

------------------------

Despite its power, template metaprogramming came to the language pretty

much by accident. Templates were never meant to be used for compile-time

algorithms and type transformations. This is why C++ punishes metaprogramming

with an insane amount of syntax noise. In CAF, we make excessive use of

templates. To keep the code readable despite all the syntax noise, we have some

extra rules for formatting metaprogramming code.

- Break `using name = ...` statements always directly after `=` if it

does not fit in one line.

- Consider the \*semantics\* of a metaprogramming function. For example,

`std::conditional` is an if-then-else construct. Hence, place the if-clause

on its own line and do the same for the two cases.

- Use one level of indentation per "open" template and place the closing `>`,

`>::type` or `>::value` on its own line. For example:

```c++

using optional\_result\_type =

typename std::conditional<

std::is\_same<result\_type, void>::value,

bool,

optional<result\_type>

>::type;

// think of it as the following (not valid C++):

auto optional\_result\_type =

conditional {

if result\_type == void

then bool

else optional<result\_type>

};

```

- Note that this is not necessary when simply defining a type alias.

When dealing with "ordinary" templates, indenting based on the position of

the opening `<` is ok, e.g.:

```c++

using response\_handle\_type = response\_handle<Subtype, message,

ResponseHandleTag>;

```

Preprocessor Macros

-------------------

- Use macros if and only if you can't get the same result by using inline

functions or proper constants.

- Macro names use the form `CAF\_<COMPONENT>\_<NAME>`.

Comments

--------

- Start Doxygen comments with `///`.

- Use Markdown instead of Doxygen formatters.

- Use `@cmd` rather than `\cmd`.

- Use `//` to define basic comments that should not be processed by Doxygen.