Zubax C++ coding conventions

This document defines the C/C++ style adopted by Zubax Robotics.

These guidelines are based on MISRA C++, High Integrity C++ Coding Standard, and the ROS C++ Style Guide. The reader is assumed to be familiar with these.

All code should respect the C++ Core Guidelines, unless doing so would contradict this document.

Contents

Naming

The general rule is to be descriptive. Don't hesitate to use longer names as they tend to improve clarity. If the purpose of an entity can't be understood without a comment or without looking at the code, consider renaming it.

Files

File names are `under_scored`. The recommended file extensions are as follows:

- C++ source files are `.cpp`.
- C++ header files are `.hpp`.
- C source files are `.c`.
- C header files are `.h`.

Types

Type names are `PascalCased`. If a type name contains an acronym, the acronym itself should be capitalized, e.g., `IOManager`. Same rule applies to `typedef` and template parameters.

In C, structs, unions, and enumerations should be referred by name rather than by tag. Suffix `_t` is not allowed. See the example:

```
// C code
typedef struct
{
    int bar;
} FooState;                // OK

typedef enum
{
    FooErrInvalidMoonPhase,
    FooErrUnmetExpectations
} FooErrors;               // OK

struct FooData_t           // Wrong
{
    double baz;
};
```

Functions

Function names should be English verbs. Usage of nouns or adjectives is prohibited, e.g. `distance()` is not allowed, but `computeDistance()` is.

Function names and method names are `camelCased`, the first character must be lowercase. If a function name contains an acronym, the acronym itself should be capitalized, e.g., `beginUSBTransfer()`.

In C++, function declarations must not use an explicit void in an empty argument list: `std::int32_t getFoo() { return 42; }`

Variables and constants

Variables and function arguments are `under_scored`. *Never use the hungarian notation.*

Function arguments that are supposed to be modified by the function should be prefixed with `out_` or `inout_`, depending on whether or not the value will be used by the function before being overwritten.

Constants, template parameters, and enumeration members are `PascalCased`. If a constant or enumeration name contains an acronym, the acronym itself should be capitalized, e.g. `USBVendorID`.

Global variables (which should be avoided unless they are absolutely necessary) and module-local variables should be prefixed with `g_`, e.g. `g_foo`.

Private and protected member variables should be suffixed with an underscore, e.g. `balalaika_`. Public member variables do not require annotation.

Preprocessor definitions

All preprocessor definitions are `CAPITALS_WITH_UNDERSCORES`. Usage of preprocessor definitions should be restricted to build configuration variables and debug macros. Never use those to define constants; use `constexpr` or `const` instead.

Namespaces

Namespace names are `under_scored`.

Scoping

In C++, all declarations except the `main()` function should reside within a namespace.

In C, all public entities should be prefixed with the name of the component they belong to (e.g. library name) in order to avoid name clashing.

```
// C code
const int FooConstant = 42;       // OK, prefixed

int fooComputeTheGreatAnswer()    // OK, prefixed
{
    return FooConstant;
}

static void doNothing()           // OK, not prefixed because not
externally visible
{
    puts("Doing nothing...");
    puts("Done.");
}
```

Templates

Never use the keyword `class` in a template argument list, use `typename` instead. The special cases where the keyword `typename` was unacceptable have been removed in C++17.

```
template <typename A, typename B>
inline decltype(std::pow(A(), B())) power(A a, B b)
{
    return std::pow(a, b);
}
```

Formatting

Indentation

Blocks of code must be indented with 4 spaces. Usage of tabs anywhere in the source code is not permitted.

The following blocks must not be indented:

- Contents of a namespace;
- Case blocks in a switch statement.

Comments are always placed on the same indentation level as the code block that contains them. There should be at least one space at the beginning of each comment line, as shown below:

```
void foo()
{
    // This is a single-line comment. Observe the space between the slash
and the first letter.
    int a = 123;  // At least two spaces are required on the left side of
the '//' sequence, unless the comment is located on a dedicated line.

    // Suppose that this is a very long comment that won't fit on one line.
    // It is preferred to use multiple single-line comments rather that a
single multi-line comment block.
    bar(a);
}
```

Declarations and definitions

Never declare more than one variable (or field) on the same line.

Asterisk and ampersand characters must be placed with the type name, not with the variable name: `std::int32_t* ptr`.

Bit field declarations should put at least one space on the right of the colon: `unsigned field: 8`.

CV qualifiers must always be placed before the type name: `const FooBar& foobar`.

Very simple member functions (e.g. getters), trivial constructors, or trivial destructors can be defined on a single line.

```
std::int32_t foo = 0;             // OK
const std::int32_t* a = nullptr;  // OK
std::int32_t& b = foo;            // OK
std::int32_t volatile *c, d;      // Wrong

class Field
{
    const std::int32_t field : 8;
public:
    Field(std::int32_t value) : field(value) { }

    std::int32_t getField() const { return field; }
};
```

Braces, brackets, parentheses

Braces should be placed on their own lines, except for single-line member functions. Never omit braces, not even for single-line statements, such as trivial `if`. Case blocks in a switch statement must be enclosed in braces as well.

Parentheses, brackets, and braces should not have spaces on the inside: `if (a) {}`, `foo[i];`.

A space should always be inserted between the `template` keyword and the following angle brace: `template <typename>`.

Operators

The following binary and ternary operators should be surrounded with spaces:

- Arithmetic `a + b`
- Logical `a || b`

- Bitwise `a & b`
- Comparison `a != b`
- Compound `a &= b`
- Ternary `a ? b : c`

Operator comma needs only one space on the right side of it: `a, b`.

Unary operators and all other operators should be adjacent to the operand, no space needed: `a[i]->b(!value, *ptr)`.

Overloaded operator definitions should not put spaces around the operator symbol: `bool operator==(const Foo&)`.

Classes, structs, unions

Visibility

Members should be grouped by visibility in the following order:

- public
- protected
- private

The ordering should be enforced using static analysis tools.

Constructor initialization list

Each member of an initialization list should be placed on a separate line. However, if the initialization list consists of one element, it can be kept on the same line.

```
class A
{
public:
    A(const std::int32_t arg_foo) : foo(arg_foo) { }

    A(const std::int32_t arg_foo, const std::int32_t arg_bar) :
        foo(arg_foo),
        bar(arg_bar)
    { }

protected:
    std::int32_t bar = 1;

private:
    std::int32_t foo = 0;
};
```

Note that some older sources use a different convention that is now deprecated, due to the fact that it could not be enforced with autoformatters.

Lines

No line of code should be longer than 120 characters.

Trailing whitespaces are not allowed; make sure to configure your text editor to automatically trim trailing whitespaces on save in the entire file.

Every text file should contain exactly one empty line at the end.

Allowed end-of-line character sequence is Unix-style (`\n`, LF). Windows-style line endings are not allowed.

Comments

All comments should be single-line comments: `//` or `///` (for documentation comments). C-style multi-line comments should not be used.

An exception applies to the case where a comment is inserted between valid tokens on the same line, such as `void foo(std::uint8_t* /* name omitted */)`. However, this pattern is discouraged.

File header

File headers should include at least the following information:

- Copyright;
- Type of the license (but not the full text of it, unless the license requires otherwise);
- Information about the author.

```
// Copyright (c) 2019 Zubax Robotics, zubax.com
// Distributed under the MIT License, available in the file LICENSE.
// Author: Pavel Kirienko <pavel.kirienko@zubax.com>
```

Coding

Variables

All variables must be default-initialized: `double foo = 0.0`. An uninitialized variable is to be considered a mistake, even if its value is to be immediately overwritten.

Variables that aren't supposed to change must be declared `const` or `constexpr`. Use `constexpr` where possible.

Variables must be defined *immediately before use*. Try to limit the scope of variables when possible (e.g., by scoping the code explicitly with braces).

It is explicitly disallowed to define all variables at the beginning of the function, the way often used with early versions of the C standard.

```
template <typename InputIter>
void packSquareMatrixImpl(const InputIter src_row_major)
{
    const std::size_t Width = CompileTimeIntSqrt<MaxSize>::Result;
// OK - const
    bool all_nans = true;
// OK - inited
    bool scalar_matrix = true;
// ditto
    bool diagonal_matrix = true;
// ditto

    {
        std::size_t index = 0;
// OK - scoped
        for (InputIter it = src_row_major; index < MaxSize; ++it, ++index)
        {
            const bool on_diagonal = (index / Width) == (index % Width);
// OK - const
            const bool nan = isNaN(*it);
// OK - const
            if (!nan)
            {
                all_nans = false;
```

```
            }
            if (!on_diagonal && !isCloseToZero(*it))
            {
                scalar_matrix = false;
                diagonal_matrix = false;
                break;
            }
            if (on_diagonal && !areClose(*it, *src_row_major))
            {
                scalar_matrix = false;
            }
        }
    }

    this->clear();

    if (!all_nans)
    {
        std::size_t index = 0;
// OK - scoped
        for (InputIter it = src_row_major; index < MaxSize; ++it, ++index)
        {
            const bool on_diagonal = (index / Width) == (index % Width); //
OK - const
            if (diagonal_matrix && !on_diagonal)
            {
                continue;
            }
            this->push_back(ValueType(*it));
            if (scalar_matrix)
            {
                break;
            }
```

```
        }
    }
}
```

Functions

All member functions must be declared `const` whenever possible (see const correctness).

Functions that return values should be annotated with `[[nodiscard]]`, excepting overloaded operators.

Type casting

Usage of the C-style cast in C++ code is strictly prohibited, with one exception of the cast to `void`, reviewed below. Use `static_cast`, `reinterpret_cast`, `const_cast`, and `dynamic_cast` instead.

It is preferable to avoid function-style cast, e.g. `FooType(x)`, unless doing so would hurt readability and conciseness. This rule should not be enforced automatically.

Cast to `void` can be used to explicitly mark a variable unused, suppressing the warnings from the compiler or the static analyzer: `(void) unused_variable;`.

Native types

Usage of `int`, `unsigned`, and all other native integral types is not permitted. Use `cstdint` instead.

Usage of `float`, `double`, `long double` is not recommended; consider aliasing them to `float32_t` and such instead.

Scoping

The statement `using namespace` should not be used, except in a very local scope (see an example below). Usage of this statement in a global scope or in a namespace scope is prohibited.

```
Vector<3> predictMeasurement() const
{
    const Scalar qw = x_[0];  // Note const
    const Scalar qx = x_[1];
    const Scalar qy = x_[2];
    const Scalar qz = x_[3];

    using namespace mathematica;  // OK - pulling the definitions of List()
and Power()
    return List(List(-2 * qw * qy + 2 * qx * qz),
                List(2 * qw * qx + 2 * qy * qz),
                List(Power(qw, 2) - Power(qx, 2) - Power(qy, 2) + Power(qz,
2)));
}
```

It is encouraged to provide comments at the ends of very long scopes (such as namespaces) that indicate what scope the closing brace belongs to. For example:

```
namespace foo
{

// A bunch of stuff here

}  // namespace foo

// Some other stuff goes here
```

Other

In C++ code, usage of the `NULL` macro or the literal zero (`0`) where a null pointer is expected is prohibited. Instead, use the `nullptr` literal.

Preprocessor

In C++, only the following uses are allowed for the preprocessor:

- Header file inclusion;
- Include guards;
- Conditional compilation (avoid unless necessary).

Any other use of the preprocessor may be allowed only if a detailed explanation and full justification of it is provided near the point of the definition.

In C++, it is explicitly prohibited to use the preprocessor for the purpose of defining a regular constant or a general-purpose macro function.

As for include guards, `#pragma once` should be preferred over the explicit set of `#ifndef ... #define ... #endif`, unless the code must be highly portable across different compilers that may not support this feature.

For conditional compilation, prefer `#if`, not `#ifdef`. Check for the existence of the variable before use is encouraged, as shown below.

```
#ifndef UAVCAN_EXCEPTIONS
# error UAVCAN_EXCEPTIONS
#endif

#if UAVCAN_EXCEPTIONS
# include <stdexcept>
#endif
```

Logging

Messages should be capitalized (do not force lowercase). Every log message should be prefixed with the name of the component where the message is emitted from, separated from the message itself with a colon and a space.

```
logMessage("TimeSyncSlave: Init failure: %i, will retry\n", res);
```

Documentation

All public entities (functions, variables, constants, etc), especially those that are parts of interfaces, should be documented in Doxygen.

Language standards

C++ projects should be written in the standard C++17 or a newer version of the language.

C code should be written in the standard C99.

It is not permitted to use any compiler-specific features, unless they are protected by appropriate conditional compilation statements. Exception applies to `#pragma once`.

Building

Compilation warnings

All release builds *must* be warning-free, unless the warnings originate from third-party components that cannot be altered.

For GCC-like compilers, the following flags are mandatory: `-Wall -Wextra -Werror`.

Other compilers should be used with equivalently strict or stricter settings.

Optimization

Aggressive compiler optimizations need to be avoided.

For GCC, optimization levels other than `-O1` should not be used. `-O2` or `-Os` should be used only in situations where the performance or memory footprint of the resulting executables are critical for the target application. `-O3` is explicitly prohibited, as well as aggressive options such as `-ffast-math -fno-math-errno -funsafe-math-optimizations -ffinite-math-only -fno-rounding-math -fno-signaling-nans -fcx-limited-range`.

Strict aliasing and strict overflow options should never be used.

Code safety assurance

> **TODO**
> Set up the list of mandatory rules from **MISRA C++** and **HIC++**; for each rule provide a list of tools that can enforce it.