devolo Software Development

C/C++ Coding Guidelines

Version 2.0

Wolfram Rösler
git@gitlab.devolo.intern:Wolfram.Roesler/software-quality.git
Last change: 17 February 2021

Table of Contents

1. Introduction

This document defines the coding guidelines for C and C++ software written at devolo. It is the product of several meetings of various C- and C++-focused development teams. The goals of this document are:

- to ensure the development of modern, high-quality source code
- to establish best practices concerning use (and non-use) of language features
- to create a common understanding of how source code is supposed to look
- to serve as a basis for discussions during code reviews
- to document a consensus rather than dictating a single person's opinion

All but the most inexperienced developers have their own coding styles and practices within which they feel most confident and productive. Forcing developers to abandon their personal styles and restrict themselves to a strictly defined coding regimen is explicitly *not* the goal of this document. Diversity is to be embraced rather than suppressed. Developers are expected to understand code written by others even if it looks different from what they would write themselves.

These are guidelines, not laws. There will always be reasons not to follow one particular guideline in one particular situation. Use common sense, avoid cargo cult.

Ignorance is no excuse for bad style. If you don't know how to use a particular language feature mentioned in the guidelines, ask (also, note that the guidelines are not a C/C++ tutorial). Others may be having the same issue, and internal training sessions are always possible. These guidelines intend to improve not only your code but also your knowledge of the C and C++ programming languages. Also, Google is your friend.[1]

Consistency is important. When editing an existing file, keep using that file's style, even if it's not the style you usually use for own code.

> *Rationale and background information is formatted like this. This information is not part of the actual coding guidelines.*

Note that some examples in this document are in C and that equivalent C++ code would look slightly different.

1 In fact, Google just pretends being your friend. You may wish to use DuckDuckGo instead. Stack Overflow is your friend, of course.

2. Basic Code Style

For existing code, keep the existing style. For new code, consider the following guidelines. In any case, style your code responsibly; being consistent and legible is more important than following a particular rule to the letter. Write code that you can justify and be proud of.

2.1 Source Code Layout

- File names: Lower case only
- File name extensions:
  - For C: `.c`, `.h`
  - For C++: `.cpp`, `.hpp`
- File encoding: UTF-8, no BOM
- Indentation: 4 spaces (no tabs)
- Line endings: Unix-style (LF)
- Maximum line length: 120 characters

> *That's 50 % more than the classical punch cards had. Horizontal scrolling isn't fun, and you want to be able to view two files side by side even if your screen is less than a meter wide.*

Brace style: Opening brace on new line in some projects, opening brace at the end of the line ("One True Brace Style") in others. On the one hand, brace style is of minor technical significance compared to other stylistic questions, and defining the brace style should not be given overly much attention in coding guidelines. On the other hand, people tend to have surprisingly intense feelings and irreconcilable opinions about it, sometimes combined with a tendency to engage in lengthy passionate discussions. If this is the case, define the brace style on a per-project basis, perhaps by the initial author or lead maintainer of the project, and use it consistently. In any case, discuss facts, not opinions.

2.2 Identifiers

Begin type names with an upper case letter. Begin variable names with a lower case letter. Begin function names with an upper or lower case letter, depending on the "importance" of the function, whatever that may mean.

If you have a function that does something similar to a standard function (for example, `size`, `empty`, `reset`, or `swap` in C++), use the standard name, don't invent your own.

Use all caps for macros and enum values (and for nothing else).

Use short, concise, but meaningful names. Use simple names for simple things. For complex identifiers, use "camel case" (`forExampleLikeThis`). Avoid underscores (`for_example_like_this`) in identifiers.

Make the length of a name roughly proportional to the size of its scope. Don't use `_t` as a postfix for type names.

> *> The C language and standard library use "snake case" (all lower case, words separated by underscores) and the `_t` postfix. By using "camel case", our own identifiers are immediately recognizable.*

Use `_` (single underscore) for variables that are never used after being defined (e. g. loop counters that are never read). Don't define any identifiers that begin with an underscore.

> *Identifiers that begin with an underscore followed by another underscore or by an upper-case letter are reserved for the C implementation (system includes, standard library, etc.), and defining them in own programs can lead to unexpected results.*

In C++, use a trailing underscore on the names of private class member variables (for example, `size_`). Don't use `m_` as a prefix.

Avoid "Hungarian notation" (that encodes the data type in the name) like the plague.

Examples:

```
// Define an enum type
typedef enum {
    RED, GREEN, BLUE
} Color;

// Define a struct type
typedef struct {
    int size;
    Color color;
    char *lpcstrName;   // I'll kill you
} Shape;

// Define a variable
Shape shape;

// Define a macro
#define SIZE (sizeof(array) / sizeof(*array))

// BAD: Avoid overly complex names for simple things
for(int index_into_my_array=0; index_into_my_array<SIZE; index_into_my_array++) {
    ...
}

// GOOD: Instead, keep the name of the loop variable short
for(int i=0; i<SIZE; i++) {
    ...
}

// Underscore indicates that the value of the variable is never used anywhere else
for(int _=0; _<10; _++) {
    DoSomething();
}
```

3. Language Features

In no particular order.

3.1 General

Feel free to use all features supported by your language level (C99, C11; C++14, C++17, etc.) Use features sensibly, don't show off. Always use the latest possible language level (e. g. there's no reason to restrict yourself to C99 if your environment supports C11).

The C language is rarely updated, however a new C++ language level is introduced every three years. Keep up with the changes, never get out of touch, never stop learning. The more you learn about a new language level the more you'll wish to use it, which is a good thing.

Stay away from anything that is “undefined behaviour” according to the C/C++ standard. Never assume that the behaviour of one platform (e. g. gcc on Linux) will be identical to that on another platform (e. g. the cross compiler for your embedded device) unless that behaviour is explicitly mandated by the language standard.

Always write "const-correct" code. Consider a missing `const` to be an error. Note that a pointer has two sides which both can be const (a pointer-to-pointer can have up to three `const`s). However, don't use `const` on function parameters or return types because it has no significance there.

Prefer "west const" (`const int a;`) over "east const" (`int const a;`) even though "east const" is more logical.

```
const long secPerDay = 24L * 60 * 60;

const char *const space = strchr(string, ' ');

// BAD: The following consts do nothing beyond causing static checker findings
const int f(const int x);

// But here the const is needed
int f(const int *x);
```

Restrict the visibility of all symbols as much as possible. Define variables as late in the code as possible, and in the innermost possible block. Avoid leaving variables uninitialized.

```
// BAD:
int count;
...
count = something();

// GOOD:
const int count = something();
```

This includes defining for loop variables inside the for loop if they aren't used outside the loop:

```
// BAD:
int i;
for (i=0; i<SIZE; i++)

// GOOD:
```

```
for (int i=0; i<SIZE; i++)
```

> *Don't worry about performance issues when the same loop variable is re-defined in several consecutive for loops, the compiler will take care of that.*

Use `for(;;)` for an endless loop. Don't use `while(true)`, `while(0==0)`, etc.

> *`for(;;)` has been the idiomatic endless loop since C was invented.*

Never put anything that generates code (function definitions) or data (variable definitions) into a header file. Use header files only for declarations, never for definitions. (On the other hand, put declarations into .c/.cpp files if they aren't needed anywhere else.)

3.2 In C

Define global functions and variables `static` if possible.

Avoid leaving undefined values in variables. Initialize them immediately when creating them, or assign something to them immediately after creating them (this frequently allows you to make them const, which is a good thing).

When specifying the size of a C string (0-terminated char array), use `+1` to explicitly indicate the terminator, for example

```
char string[16+1];    // String can hold 16 characters plus the terminator
```

Avoid repeating size specifications, even if they are encapsulated in a macro. Use the (devolo-defined) `COUNT_OF` macro to get the number of elements in an array, and remember that `sizeof(char)` is 1 by definition. For example:

```
#include <devolo_defs.h>     // from devolo-shared, defines COUNT_OF

#define LENGTH (32+1)
char data[LENGTH];

// BAD: Don't repeat the size
memset(data, 0xFF, LENGTH);

// BAD: No need to multiply by 1
memset(data, 0xFF, sizeof(char) * LENGTH);

// GOOD: Instead simply use sizeof
memset(data, 0xFF, sizeof(data));

int values[100];

// BAD: Don't repeat the type or size
fread(values, sizeof(int), 100, stdin);

// GOOD: Instead use this:
fread(values, sizeof(*values), COUNT_OF(values), stdin);
```

> *Note that `COUNT_OF` works on arrays only, not on pointers.*

Avoid magic numbers in code. Most of all, avoid repeating magic numbers. Encapsulate magic numbers in a #define symbol or (if appropriate) a const variable.

Use `NULL`, not `0` (with or without cast), as a null pointer literal:

```
// BAD
char *p = 0;

// BAD
char *p = (char*)0;

// GOOD
char *p = NULL;
```

However, for best portability, `(char*)0` is required instead of `NULL` in varargs functions, like

```
printf("%p", (char*)0);
```

> *The reason for this is that NULL may be defined simply as 0, and pointers and integers may have different sizes.*

Don't assume null pointers to be all-bits-zero. They usually are, but you can't rely on it in portable code.

```
struct {
    int a;
    char *p;
} buf;

// This may or may not set buf.p to NULL:
memset(&buf, 0, sizeof(buf));

// Only this does:
buf.p = NULL;
```

Generally, assume that any pointer is NULL unless you explicitly make sure it isn't.

Use casts only if necessary, rely on implicit conversions if possible. Remember that all pointers implicitly convert to and from `void*`:

```
// BAD: The casts aren't needed
char *const data = (char*)malloc(SIZE);
memset((void*)data, 0, SIZE);

// GOOD
char *const data = malloc(SIZE);
memset(data, 0, SIZE);
```

3.3 In C++

Note that most topics of the C chapter apply to C++ as well if the respective language features are used in a C++ program. However, prefer using C++ features over C features whenever possible. C++ is not just “C with classes” but a completely different programming language that follows its own paradigms and philosophy; don’t write C-style in a C++ program just because one language is a subset of the other.

3.3.1 General

Use the "modern C++" style introduced with C++11.

If you can use `constexpr` instead of const, do so.

> *`constexpr` guarantees that the value is evaluated at compile time. `constexpr` variables don't exist at runtime.*

Prefer AAA ("almost always auto") style:

```
// BAD
Type x(1, 2, 3);

// GOOD
auto x = Type(1, 2, 3);
```

> *Herb Sutter's article on AAA, linked in the appendix, explains in great detail why this style is a good idea.*

Use "range for" whenever possible. For example:

```
// BAD: Old style
for(auto it=container.begin(); it!=container.end(); ++it) ...;

// GOOD: Modern C++
for(const auto& v : container) ...;

// Very useful with ad-hoc initializer list
for(const auto n : { 1, 5, 13, 99 }) ...;
```

Restrict the visibility of all symbols as much as possible. Don't make something public if it can be private, and define variables as late in the code as possible (in the innermost possible block). Use "RAII style" to initialize variables immediately when defining them. Design your classes to allow RAII usage. Example:

```
// BAD: Separate construction and initialization
Shape shape;
int size;
shape.Init(1, 2, 3);
size = shape.size();

// GOOD: Resource acquisition is initialization
auto shape = Shape(1, 2, 3);
auto size = shape.size();
```

> *With RAII, the object is never in an uninitialized state.*

Avoid returning status values ("0 if ok, else error number"), use exceptions instead (but use exceptions only for exceptional/error situations, not to return regular results). If you need to create an own exception class, derive it from `std::exception` (however, in most cases you can simply use `std::runtime_error` or one of the other standard exception classes). Never throw from a destructor.

Use `using` instead of `typedef`. Example:

```
// BAD
typedef std::map<int, Shape> ShapeMap;
```

```
// GOOD
using ShapeMap = std::map<int, Shape>;
```

Use standard library containers whenever possible. The default container is `std::vector`; use other containers only if there's a reason to do so. Prefer `std::unordered_map` over `std::map`, `std::unordered_set` over `std::set`, and `std::forward_list` over `std::list` if possible.

> *Although `vector` may require reallocation when new elements are added, it does exactly what modern processors are good at. `list` may mess up the CPU cache which comes with a huge performance penalty. `unordered_map` is usually a hash map (O(1) lookup in average case) while `map` is a binary tree (O(log n) lookup plus balancing). `forward_list` is a singly linked list, `list` is doubly-linked.*

Avoid C-style strings (char arrays terminated with `'\0'`), use `std::string` instead (but don't forget that string literals still give you a C-style string unless you use the `s` postfix).

Avoid C-style (aka raw) arrays. If the array size is known at compile-time, use `std::array`, else use `std::vector` or another suitable container. Note that `std::string` is perfectly fine for binary data (including, but not limited to, UTF-8). Example:

```
// BAD
int array[10];

// GOOD
std:array<int, 10> array;

// BETTER, because AAA
auto array = std::array<int, 10>();
```

> *`std::array` is essentially type-safe syntactic sugar for raw arrays, and will compile into exactly the same machine code.*

Use standard library algorithms whenever possible. For example, if you want to know if a vector contains an element with particular properties, don't manually iterate over the vector, but use `std::any_of` instead.

Use references when possible, and (raw[2]) pointers when necessary. Expect any pointer you receive to potentially be NULL (but don't use `NULL` but `nullptr`). In general, try to avoid pointers.

Never use C-style casts in a C++ program. Use C++ casts (`static_cast`, `const_cast`, `reinterpret_cast`, etc.) instead.

Usually you want to avoid `std::thread` and use `std::async` instead (or a thread pool, see link in the appendix). Either way, always write thread-safe code, even if your application is not (yet) multi-threaded.

> *`std::thread` cannot return a result or forward an exception to the caller, and `aborts` in the destructor unless explicitly `joined` or `detached`.*

2 C-style pointers, as opposed to smart pointers.

Use `std::atomic` for variables that are used concurrently in more than one thread. Don't use `volatile` for this purpose, it's something completely different.

> *`volatile` prevents access optimizations to memory that may change outside the program's control, e. g. addresses bound to hardware registers.*

In derived classes, always use `override` when overriding virtual functions.

Declare all single-parameter constructors `explicit` unless you really want implicit type conversion (you usually don't).

> *When C++ was invented, it was thought that implicit type conversions were a good idea, which is why it is the default. It was later found out that implicit conversions cause more problems than they solve, which is the reason why, for example, `std::string` doesn't implicitly convert to a C string.*

Obey the "rule of five" (if you need to implement your own copy constructor, move constructor, assignment operator, move assignment operator, or destructor, you usually need to implement all five).

> *Before C++11 and move semantics, that used to be the "rule of three".*

In a class definition, initialize member variables in the variable definition instead of in the constructor. Example:

```
// BAD
class A {
public:
    int n;
    A() : n(0) {}
};

// GOOD
class B {
public:
    int n = 0;
}

// You're going to regret this
class C {
public:
    int n;
};
```

Avoid `std::bind`, use a lambda instead. It's practically always possible.

> *`std::bind` is an ugly crutch that tried to put lambda functionality into a language that doesn't have lambdas. As of C++11, we can do better.*

Avoid C file handling functions (`FILE*`), use I/O streams instead.

3.3.2 Namespaces

When writing a library, put all globally visible functions and variables (i. e., everything that's exported from the library) into a namespace.

Use an anonymous namespace for global functions and variables that are to be visible within one file only. Don't use `static` for this purpose. Example:

```
// BAD
static int somevalue = 0;
static void Something() {
    ...
}

// GOOD
namespace {
    int somevalue = 0;
    void Something() {
        ...
    }
}
```

> *static does too many different things in one keyword.*

Writing `using namespace` to save typing isn't as useful as it sounds. Never put `using namespace` into a header file, and never put `using namespace std;` anywhere. Spell out namespaces all the time, even it it's tedious. Example:

```
// BAD
using namespace std;
using namespace boost::asio::ssl;
using namespace boost::asio::ssl::context;

vector<string> v;
auto ctx = context(sslv23);
ctx.set_options(default_workarounds | no_sslv2 | single_dh_use);

// GOOD
std::string<std::vector> v;
auto ctx = boost::asio::ssl::context(boost::asio::ssl::context::sslv23);
ctx.set_options(
    boost::asio::ssl::context::default_workarounds |
    boost::asio::ssl::context::no_sslv2 |
    boost::asio::ssl::context::single_dh_use);
```

3.3.3 Resource Management

Use smart pointers instead of raw pointers whenever possible. Avoid invoking `new`/`delete` directly, use `std::make_unique` or `std::make_shared` instead. Never use `malloc`/`free` in a C++ program. Remember that `std::unique_ptr` has no runtime overhead compared to a raw pointer.

```
// BAD
Type* p = new Type(1, 2, 3);
...
delete p;

// GOOD
auto p = std::unique_ptr<Type>(new Type(1, 2, 3));
```

```
// BETTER
auto p = std::make_unique<Type>(1, 2, 3);
```

Use `std::unique_ptr` if possible and `std::shared_ptr` if necessary. Avoid passing `std::shared_ptr` to functions by value, use a reference instead:

```
// BAD
void f(std::shared_ptr<Type>);

// GOOD
void f(std::shared_ptr<Type>&);

// ALSO GOOD
void f(Type&);
```

Don't use `std::auto_ptr`, it's an abomination (and has been removed in C++17).

Avoid manual resource management. That is, never manually invoke a de-allocation function, but use a destructor or scope guard to do that for you. When interfacing with a C API, unless you have a dedicated C++ wrapper class, Boost scope guards are an easy way to make sure resources are reliably de-allocated:

```
// If you get something from a C API that requires manual de-allocation ...
const auto handle = allocate_something();

// ... don't de-allocate manually
deallocate_something(handle); // don't do that

// Instead use a scope guard
BOOST_SCOPE_EXIT_ALL(handle){ deallocate_something(handle); };
```

If you're not using Boost, put something equivalent to `BOOST_SCOPE_EXIT_ALL` into a central header file and use it consistently, for example:

```
#include <functional>

class ScopeGuard {
public:
    explicit ScopeGuard(std::function<void()> f) : f_(f) {}
    ~ScopeGuard() { f_(); }

private:
    const std::function<void()> f_;
};

const auto handle = allocate_something();

const auto _ = ScopeGuard([&handle](){ deallocate_something(handle); });
```

If the handle is a pointer, you can also use a smart pointer with a custom deallocator:

```
const auto handle = std::unique_ptr<Handle,void(*)(Handle*)>(
    allocate_something(),
    deallocate_something
);
```

4. Data Types

In C, put the pointer symbol with the name (`char *p`). In C++, put the pointer and reference symbol with the type (`char* p`, `int& n`).

> *`char *p` is the classical C style while `char* p` is the classical C++ style.*

Use `int` as the default integer type if you don't need more than 16 bits, and `long` if you don't need more than 32 bits. If the exact number of bits is important, use one of the fixed-width integer types (`int32_t`, `uint8_t` etc.) Don't use `short`, `long long`, or home-grown aliases (`INT8U` or whatever) unless you really have to (e. g. because they are required by an API you have no control over).

> *int is the data type the CPU can handle best. The C standard guarantees at least 16 bits for int and at least 32 bits for long. Although desktop platforms usually use 32-bit ints nowadays, int is 16 bits on at least one of our embedded platforms. uint8_t guarantees certain overflow and packing behaviour but will save neither memory (because the value will occupy a whole word or CPU register anyway, unless in an array) nor code (it may actually require more code due to worst-case assumptions).*
>
> *`#include <stdint.h>` is required for the fixed-width integer types.*

Use `unsigned` only if necessary. Frequently it's not necessary (it sometimes is, however, because `size_t` is unsigned, unfortunately.) For example, don't use `unsigned` for numbers that fit in 15 bits even if they can never be negative (use `int` instead).

> *There's also `ssize_t`, a signed alternative to `size_t`, but it's a Posix extension, not standard C or C++. C++20 has `ssize()` which is the same as `size()` but returns a signed value, to remedy what's today considered a design error.*

Use `char` only for characters, not as a small integer type. Remember that `char` may be signed or unsigned.

Use `bool` for booleans. Don't use `int` (because it works), `char` (because it's small), `BOOL`, or any other home-grown data type for boolean values.

> *In C, `#include <stdbool.h>` is required for `bool`.*

Use postfix notation (e. g. `0L`, `1u`) instead of casts (e. g. `(long)0`, `(unsigned)1`) if you need long and/or unsigned literals. For floating-point literals, use a decimal point (e. g. `0.0`) instead of a cast (`(double)0`).

Don't use `assert`. It may do nothing (not even test the assertion), it may abort your program, both of which are horrible ways of dealing with error conditions. Feel free to use `static_assert`, though.

> *`static_assert` is tested at compile-time. If the assertion fails, the build fails.*

5. Preprocessor

5.1 General

In `#include`, put the file name in double quotes (`"..."`) if the include file is in the current directory (=the same directory as the source file that includes it), or is specified by a path relative to the current directory. Use angle brackets (`<...>`) otherwise. For example:

```
#include <dsl_coexistence.h>                 // Somewhere on the search path
#include "dsl_coexistence_internal.h"        // In the current directory
#include "../../dsl_coexistence/mockup.h"    // Relative to current directory
```

> *The difference is that the compiler searches `<>` on the include search path (e. g. compiler option `-I`), and `""` in the current directory plus on the include search path. The convention "double quotes if we wrote it, angle brackets if the compiler maker wrote it" is misleading and doesn't reflect the intention or definition of the C/C++ language.*

Don't tell the compiler to search the current directory for include files (i. e. don't use `-I.`), that's what the double quotes are for.

5.2 In C

Always put macro arguments in parentheses.

> *You'll get static checker warnings if you don't.*

Prefer enums, variables and functions (which are scoped and typed) over macros.

> *The compiler has the freedom to optimize variables away entirely under certain circumstances, especially if they are const, so there's no performance or code size disadvantage over a #define symbol. Also, functions are guaranteed to evaluate each argument exactly once.*

Use the "do-while(0)" technique to safely wrap multiple statements in a macro:

```
#define RESET(x) \
    do { \
        memset(&(x), 0, sizeof(x)); \
        (x).initialized = true; \
    } while(0)
```

> *This won't mess up the syntax even in contexts like `if (something) RESET(var1); else RESET(var2);` or other edge cases. The loop is optimized away by the compiler.*

Never put a `;` at the end of a macro:

```
// BAD
#define RESET(x) ((x).initialized = false);

// GOOD
#define RESET(x) ((x).initialized = false)
```

5.3 In C++

In C++, avoid `#define` whenever possible (and that means, essentially always). Use C++ language features (enums, constexpr variables, template functions, etc.) instead. Example:

```
// Avoid:
#define RED 0
#define GREEN 1
#define BLUE 2

// Use instead:
enum class Color { Red, Green, Blue };

// Avoid:
#define SECONDS_PER_MINUTE 60
#define HOURS_PER_DAY 24

// Use instead:
constexpr auto secondsPerMinute = 60;
constexpr auto hoursPerDay = 24;

// Avoid:
#define DIV(a,b) ((b)==0 ? 0 : (a) / (b))

// Use instead:
template<typename T>
T Div(T a,T b) {
    return b==T(0) ? T(0) : a / b;
}
```

6. Optimizations

Write the code you want to read and leave the rest to the compiler. Seriously, compilers do an amazing job at optimizing code; you won't improve much if you try to do their work. If you don't believe that, check your code in Compiler Explorer.

> *Sadly, that may not always be true for the 15-years-old vendor-specific cross compiler for your embedded platform.*
>
> *The compiler's freedom to do whatever it wants if code exhibits "undefined behaviour", especially if it serves aggressive optimizations, is one reason to avoid everything "undefined", even if "it should work" according to what you believe to be common sense.*

Remember that even $O(n^2)$ isn't too bad if n is small. Remember that premature optimization is the root of all evil.

Don't take the above as a justification for writing stupid code, though.

Don't guess how code performs because you will usually guess wrong. If you think you need to optimize something, measure first. Remember that not knowing how your program performs is the *real* root of all evil.

7. Embedded C++ Considerations

Due to restricted resources, the following C++ features may be unavailable or undesirable on embedded systems:

- Exceptions
- Dynamic memory (new, delete)
- Standard library containers (`std::string`, `std::vector`, etc.)
- Run-time type identification (including `dynamic_cast`)
- Virtual functions
- Threads
- Futures and promises
- C++17 file system functions
- Some parts of the Boost library

However, the following C++ features can be used without runtime penalty even on embedded systems:

- Constructors and destructors (including RAII and scope guards)
- Member functions (unless virtual)
- Member variable initializers
- References instead of pointers
- Templates
- Function overloading
- Function default parameters
- Move semantics
- Namespaces
- C++ casts
- `std::array`
- `std::unique_ptr` (if using dynamic memory is acceptable)
- Standard library algorithms
- Syntactic sugar (range for, auto, etc.)
- Some other parts of the Boost library

The following C++ features may even improve performance compared to a classical C-style implementation:

- Templates
- `constexpr` variables
- `constexpr` functions
- `constexpr if` (C++17)

> *All of these guarantee that certain things happen at compile time that would (or could) otherwise happen at runtime.*

When in doubt about the cost of a language feature, consult Compiler Explorer.

8. Comments

Comment your code thoroughly in complete English sentences. Everybody reading it, including yourself in a few months, will be very grateful.

Let the code itself explain the "how", use comments to explain the "why". Don't state the obvious, but document everything that may not be obvious to a future reader whose background is different from yours.

Keep comments understandable, yet concise. Avoid jargon or acronyms the reader may be unfamiliar with. Avoid witty, humorous, or ironical comments; someone trying to fix your code in a stressful situation will not be in the mood to laugh about smart-ass jokes.

When you modify code, don't forget to also modify the comments. "Comment rot" is a bad thing (not bad enough to justify writing fewer comments, though). Remember, "if code and comments disagree, both are probably wrong".

Keep comments local (referring to the code that immediately follows the comment). Avoid commenting far-away code locations; such comments are usually forgotten when the code changes.

Begin every file with a short description and authorship/copyright notice, for example

```
/*
 * devolo utility library
 * Helper functions for unit tests
 * by Wolfram Rösler 2020-03-26
 * © devolo AG
 */
```

Begin every function with a header comment that describes exactly

- what the function does
- what the parameters do
- what the return values are
- preconditions and postconditions
- side effects and dependencies (e. g. on global variables)
- transfer of ownership (e. g. when returning a pointer that the caller must free)
- cross-references to related functions
- bugs and deficiencies
- anything that may be unknown or surprising to an unsuspecting reader

Generally, this function header comment is placed in the source file where the function is implemented, not in the header file where it's declared. You may add a single-line comment that very briefly summarizes what the function is about to the header file. In special cases (e. g. interfaces with

multiple implementations, or libraries with few implementers and many independent users), these rules may be reversed (full documentation in the header file, one-liner in the source file).

> *See appendix for a discussion of the pros and cons of the two solutions.*

As a guideline, it should be possible – at least in theory – to re-write the function from scratch after reading the header comment, without reading the actual code.

Use "Doxygen style" for the function header comment. Example:

```
/**
 * Return the last message that was submitted to the queue.
 *
 * If no message exists in the queue, waits until timeout for a
 * message to arrive.
 *
 * Messages longer than size will be silently truncated.
 *
 * @param ctx Pointer to the message queue context.
 * @param[out] data The message is copied into this buffer.
 * @param size Size of the buffer at data.
 * @param to When to return if no message is received. May be NULL (=no timeout).
 *
 * @returns 0 if dequeued successfully, -1 if error.
 *
 * If an error occurs, -1 is returned, and errno is set to
 * one of the following:
 *
 *      - EINVAL: context and/or data is NULL
 *      - ETIMEDOUT: No data was received within until timeout
 *
 * @see dvl_queuePush
 */
int dvl_queuePop(dvlQueue *ctx,void *data, size_t size, const struct timespec *to)
{
    ...
}
```

Using `@brief` is optional (use it if the brief description, which is in the first line of the comment, consists of multiple sentences or paragraphs). Use `@param[out]` and `@param[in,out]` where appropriate. Don't use `@param[in]`.

Also use header comments for data types (`class`, `struct`, `enum`).

Except for those Doxygen-style comments, use C++-style comments (`//`).

In your Doxyfile, set the following variables to abort compilation if a function is not properly documented:

```
WARN_IF_DOC_ERROR = YES
WARN_NO_PARAMDOC = YES
WARN_AS_ERROR = YES
```

Comments are not just decoration. Before you modify code, make sure to actually read them.

9. Compiler Settings

Always compile at maximum warning level. Enable as many warnings as possible, and explicitly disable the few warnings that definitely make no sense. Set the compiler to treat warnings as errors (that is, abort compilation if there are any warnings), at least for developer builds.

> *When code is built across a variety of platforms and compiler versions, it may not always be possible to guarantee a warning-free build on all of them.*

Compile on the highest possible optimization level (possible for your project, not for the compiler).

10. Stylistic Preferences

The following are largely matters of personal preference rather than of technical reasoning. You may or may not agree. Avoid religious discussions about such topics.

10.1 General

Use `long` and `unsigned` as if they were real data types and not just modifiers. Don't write `long int` or `unsigned int`.

Avoid extraneous parentheses. For example:

```
// BAD
return ((x) ? (f(x)) : (g(x+1) * 2));

// GOOD
return x ? f(x) : g(x+1) * 2;

// BAD: return is not a function
return(0);

// But treat sizeof like a function:
size_t x = sizeof(int);
```

> *Parentheses are a poor excuse for not knowing about operator precedence.*

Don't explicitly compare booleans to `true` or `false`, they are boolean already. Also, use pointers and characters in a boolean context without explicit comparison to `NULL` or `'\0'`. For example:

```
bool f = something();
char *p = somethingElse();

// BAD
if (f==true) ...;
if (f==false) ...;
if (p!=NULL) ...;
if (p==NULL) ...;
if (*p!='\0') ...;
if (*p=='\0') ...;

// GOOD
if (f) ...;
if (!f) ...;
if (p) ...;
if (!p) ...;
if (*p) ...;
if (!*p) ...;
```

> *If f is bool and you write if (f==true) because you prefer to explicitly compare the bool to a value, remember that f==true is also bool, so why not write if ((f==true)==true)? And why stop there?*

In if-else, avoid negation in the "if" part. For example:

```
// BAD: More difficult to understand
if (!allowed) {
    reject();
} else {
    accept();
}

// GOOD: Easier to understand
if (allowed) {
    accept();
} else {
    reject();
}
```

This also applies to the preprocessor (avoid `#ifndef ... #else`).

Prefer `switch` over if-else chains.

Avoid extraneous white space. Especially, avoid spaces after opening and before closing parentheses or brackets (that is, make it look like normal English). Example:

```
// BAD
for ( int i = 0 ; i < 10 ; i ++ ) {
    f( a[ i ] );
}

// GOOD
for (int i=0; i<10; i++) {
    f(a[i]);
}
```

Don't use "Yoda conditions":

```
// BAD
if (0==value) ...;

// GOOD
if (value==0) ...;
```

> *Yoda conditions don't feel like natural language (e. g. you say "if the food is tasty", not "if tasty is the food", unless you are Yoda). They are intended to catch errors like `if (value=0)` but modern compilers will warn about those anyway.*

Note that the following is *not* a Yoda condition, so feel free to use it if appropriate:

```
if (0 < value && value < 100) ...;
```

Expect any pointer to be NULL unless you've verified that it's not. However, `free(NULL)` is defined to be a no-op, so there's no need to test a pointer against NULL before freeing it.

If possible, use `#pragma once` as an include guard (at the beginning of all include files). Don't use `#ifndef`/`#define`/`#endif` with a hopefully-unique name.

> *That name won't become more unique by beginning or ending it with underscores. `#pragma once` is non-standard but supported by all modern compilers.*

Collect declarations in header files that are organized according to a "module" or "public/private interface" approach. There's no need to create a corresponding .h file for every .c file.

Avoid `goto`. It's so 1970s.

10.2 In C

You can't pass an array to a function, all you can pass is a pointer. So, to avoid confusion, don't declare function arguments as arrays. For example:

```
// BAD: data is not an array (e. g. COUNT_OF(data) won't work as expected)
void f(int data[]);

// BAD: that may or may not be the actual size, COUNT_OF(data) won't give you 10
void f(int data[10]);

// GOOD: data is a pointer, so declare it as such
void f(int *data);
```

10.3 In C++

In class definitions, put the `public` part first, then the `protected` part, finally the `private` part.

Avoid the "uniform initialization syntax" (if possible – it's not always possible) because it's ugly:

```
// BAD
int a{123};

// GOOD
int a = 123;

// BETTER
auto a = 123;
```

For function return parameters ("call by reference"), prefer references to pointers. Better still, don't use a return parameter but return a `std::pair` or `std::tuple`. Remember that, as of C++11, there's no need to be afraid of returning objects from functions for performance reasons (as long as move constructors are supplied, which they should be). Example:

```
// BAD
int FunctionReturningTwoNumbers(int*);

// GOOD
int FunctionReturningTwoNumbers(int&);

// BETTER
std::pair<int,int> FunctionReturningTwoNumbers();
```

In general, prefer "value semantics" over "reference semantics".

Avoid defining member functions inside the class definition. Instead, declare the function in the class definition (usually in a .hpp file) and define it separately (in a .cpp file). Use inline functions only if they are very trivial (e. g. simple getters/setters).

Don't use `noexcept`. It's less useful than it sounds.

> *`noexcept` means "this function never throws" in the sense of "it may throw but if it does then the program will be `aborted`".*

Use the classical logical operators (`&&`, `||`, etc.) instead of their keyword variants (`and`, `or`, etc.)

Don't use an operating system API for anything that can be done with standard C++. For example, use `std::async` instead of pthreads for multithreading, and (in C++17) use `std::filesystem` instead of `boost::filesystem` or Unix system calls (`stat`, `opendir`, etc.)

11. 3rd Party Libraries

Before implementing complex general-purpose algorithms, check if an open-source implementation exists that could be used instead. However, remember that third-party components are a liability as well as an asset, and not everything on GitHub or StackOverflow is suitable for production use. Avoid open-source libraries that aren't used widely, haven't achieved sufficient maturity, or offer significantly more functionality than required.

> *For example, don't include a complete web server library into your project if all you need is the URI parser.*

Always check the license before using a 3rd party library. For example, the MIT license is usually acceptable while GPL may be not.

> *GPL is contagious: If a program uses GPL-licensed components then it must also be GPL-licensed, which among other things means that its source code must be available to the end user.*

Always keep your libraries up-to-date. Update them to the latest version once in a while, you don't want to get caught in the "legacy dependency trap". Having good unit test coverage helps validating that everything still works after a library update.

When fixing or extending third-party libraries, make the changes available to the original author (e. g. through a GitHub pull request). This is how we pay back to the Open Source community.

12. Quality Assurance

Use a "test-driven development" (TDD) process. Write unit tests first, then write code until the unit tests pass, finally refactor the code.

Before fixing a bug, first reproduce it in a unit test case.

Unless you are refactoring, never change code unless it's to make a failing test pass.

Don't hesitate to invest a lot of time into your unit tests, it will pay out sooner than you think.

> *During strict test-driven development, it's not uncommon for unit tests to take longer to write than the actual code.*

Use code coverage measurement tools to find which part of your code isn't touched by unit tests yet. However, don't tailor your code for a high coverage percentage.

> *Coverage percentage is not a quality goal. It's rather meaningless actually.*

Make your code pass static analysis cleanly without findings.

> *The CI pipeline will fail if it doesn't.*

Use dynamic analysis (ASAN etc.) to check for leaks or memory access errors.

Fuzz-test all code that processes external input (e. g. data received from the network).

For more information, refer to the devolo Software Quality White Paper.

13. Appendix

13.1 Resources

Address Sanitizer & friends: https://github.com/google/sanitizers

Assorted code snippets and goodies: https://gitlab.com/wolframroesler/snippets

C++11 and C++14: https://wiki.devolo.intern/index.php/C%2B%2B11_And_C%2B%2B14

C11: https://en.wikipedia.org/wiki/C11_(C_standard_revision)

C++17 front-end for sqlite: https://gitlab.devolo.intern/SG/dbapi

C99: https://en.wikipedia.org/wiki/C99

C++ Core Guidelines: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

C language reference: https://en.cppreference.com/w/c/language

C++ language reference: https://en.cppreference.com/w/, http://www.cplusplus.com/

Compiler Explorer: https://godbolt.org/

C++ profiler: https://gitlab.com/wolframroesler/profiling

Doxygen keywords: http://doxygen.nl/manual/commands.html

Fixed-width integer types: https://en.cppreference.com/w/cpp/types/integer

Identifier styles eye tracking study: http://www.cs.kent.edu/~jmaletic/papers/ICPC2010-CamelCaseUnderScoreClouds.pdf

Introduction to static code analysis tools: https://wiki.devolo.intern/upload/a/ad/static-analysis.pdf

Linux kernel coding style: https://www.kernel.org/doc/html/v5.1/process/coding-style.html

List current version of various 3rd party libraries: https://gitlab.devolo.intern/SG/sg-productionDB/blob/develop/3rdparty/latest.sh

Modern C++: https://herbsutter.com/elements-of-modern-c-style/

One True Brace Style: https://en.wikipedia.org/wiki/Indentation_style#K&R_style

Online compiler to try out and run your code: https://wandbox.org/

Software Quality White Paper: <to be added>

Standard library algorithms: http://www.cplusplus.com/reference/algorithm/

Test driven development: https://gitlab.com/wolframroesler/Talks/blob/master/tdd.pdf

Thread pool and threading tutorial: https://gitlab.com/wolframroesler/ThreadPool

Value vs. reference semantics: https://isocpp.org/wiki/faq/value-vs-ref-semantics

West const vs. east const: https://gitlab.com/wolframroesler/Talks/blob/master/const.pdf

13.2 Pros and cons: Function header comments in .c or .h files?

Reasons to put the comments into the .h files:

- Clean separation of code and interface
- Header files are more easily accessible for modules like devolo-shared
- Comments are in a single place only for functions that have more than one implementation

Reasons to put the comments into the .c files:

- They need to be in the .c files for static functions anyway
- Keeps comments as close to the code as possible
- Less comment rot because the comment is in plain sight when modifying the code
- Simplifies converting functions from static to public and vice-versa
- Only a single file needs to be changed if the prototype remains the same (may prevent merge conflicts)
- GitLab MRs show .c files before .h files of the same name, so comments appear before the implementation, not after